MUDHALVAN PROJECT

MERN stack powered by MongoDB

PROJECT TITLE

<u>SB FOOD APPLICATION using MERN Stack</u>

Submitted by the team members of Final Year IT ‘B’

MONIKA.L	31 421205055
NAVYA.P	31 421205058
PRADEEPA.A	311421205061
SURYAKALA.P	31 421205088

DEPARTMENT OF INFORMATION TECHNOLOGY

MEENAKSHI COLLEGE OF ENGINEERING

12, VEMBULIAMMAN KOVIL STREET, WEST K.K. NAGAR,

CHENNAI - 600 078, TAMILNADU.

(AFFILATED TO ANNA UNIVERSITY)

SB Food Application Using MERN Stack

ABSTRACT:

The SB Food Application is a comprehensive application for food application leveraging the MERN stack (MongoDB, Express, React, Node.js) to provide accessible, flexible, and user-friendly learning opportunities for customers worldwide. SB FOOD APP incorporates a suite of features designed to streamline online ordering, including interactive tools, and flexible accessibility across devices.

Through a client-server architecture, SB food app enables real-time data exchange and supports robust functionalities such as user authentication, role-based access, and payment processing for premium content. By offering a seamless and engaging user experience, this platform empowers customers to order their craved food at anywhere, interact with peers and owners for customised orders, and receive delivery on time even anytime.

The platform also supports hotel owners in creating and managing menu items and enables customers to oversee the system effectively. This documentation details the requirements, architecture, features and implementation steps necessary to develop the SB food application using MERN, serving as a foundational guide for an effective online food ordering ecosystem.

Keywords: Food Application, Web Application, MERN Stack, Mongo DB

TABLE OF CONTENTS

1. INTRODUCTION:

1.1. PROJECT TITLE:

SB FOOD APPLICATION Using MERN Stack

1.2. TEAM MEMBERS & THEIR ROLE:

▶ SURYAKALA P - Role: Project Lead & Front end Developer, responsible forcoordinating the team, overseeing project milestones, and ensuring timely completion of deliverables and also focusing on designing & implementing the user interface using React & Material UI.

▶ NAVYA P - Role: Frontend Developer, focusing on designing & implementing the user interface using React & Material UI for an engaging and intuitive user experience.

▶ MONIKA L - Role: Backend Developer, responsible for building RESTful APIs, managing server-side functionality & ensuring data security using Node.js & Express.

▶ PRADEEPA A - Role: Database Administrator, in charge of managing of the MongoDB database, handling data storage& ensuring efficient data retrieval & integrity.

2. PROJECT OVERVIEW:

In recent years, the demand for Food ordering at late has surged, driven by the need for flexible, accessible, and diverse le options that cater to different schedules, locations, and Taste preferences. An FOOD ORDERING is online system designed to meet this demand, providing a comprehensive food ordering environment that a lows users to engage with content, track location, and live interaction with owners for better and customised delivery. Built using the MERN stack, leverages MongoDB for data storage, Express.js for server-side functionality, React for a dynamic front-end interface, and Node.js for back-end development, ensuring a scalable, efficient, and responsive delivery platform.

SB food ordering is built with a focus on accessibility and interactivity, allowing customers of all backgrounds and technical proficiency to navigate the platform easily. It supports features like friendly registration, discussion forums, live chats, and progress tracking. Moreover, the platform offers flexibility for both free and paid access, catering to broad customers.

2.1. PURPOSE:

The purpose of the SB FOOD APPLICATION is:

- Customer Convenience:

Easy browsing and ordering from a variety of restaurants and cuisines.
Real-time tracking of orders and delivery status.
Multiple payment options and offers to enhance user satisfaction.

Restaurant Empowerment:

- Facilitate restaurant partnerships by providing tools to manage menus, orders, and customer feedback.
- Increase visibility and reach of restaurants to a broader customer base.

Streamlined Delivery:

- Provide delivery partners with an intuitive system to accept orders, navigate to destinations, and manage delivery schedules.
- Real-time routing and optimization for efficient delivery.

Operational Optimization:

- Utilize data and analytics to predict demand, reduce delivery times, and improve customer satisfaction.
- Automated workflows for order processing, payments, and notifications.

Revenue Generation:

- Offer commission-based partnerships with restaurants and delivery agents.
- Monetize through ads, premium listings, and subscription plans.

Enhanced Community Engagement:

- o Foster trust and loyalty with user reviews, ratings, and personalized recommendations.
- o Support local businesses by promoting neighborhood restaurants.

2.2. FEATURES:

The feature of the Sb food application encompasses the development, deployment, and maintenance of a comprehensive food delivery application using the MERN stack.

Key components and features within this feature includes:

- User Management: Support user roles for customers, owners, and admin, including registration, login, and profile management.

Restaurant Discovery:

- Payment and Subscription Options: Provide a payment gateway for paid courses, offering both one-time purchases and subscription models.
- Cross-Device Accessibility: Ensure compatibility across various devices (PCs, tablets, smartphones) to allow access from any location with an internet connection.
- Front-end & Back-end Integration: Leverage MERN stack for efficient front-end and back-end communication and database management.
- Admin Panel: Include an administrative dashboard for monitoring user activity, managing course listings, and handling platform maintenance tasks.

3. SYSTEM REQUIRMENTS:

3.1. HARDWARE:

Operating System: Windows 8 or higher

RAM: 4 GB or more (8 GB recommended for smooth development experience)

3.2. SOFTWARE:

Node.js: LTS version for back-end and front-end development

MongoDB: For database management using MongoDB Atlas or a local instance

React: For front-end framework

Express.js: For back-end framework

Git: Version control

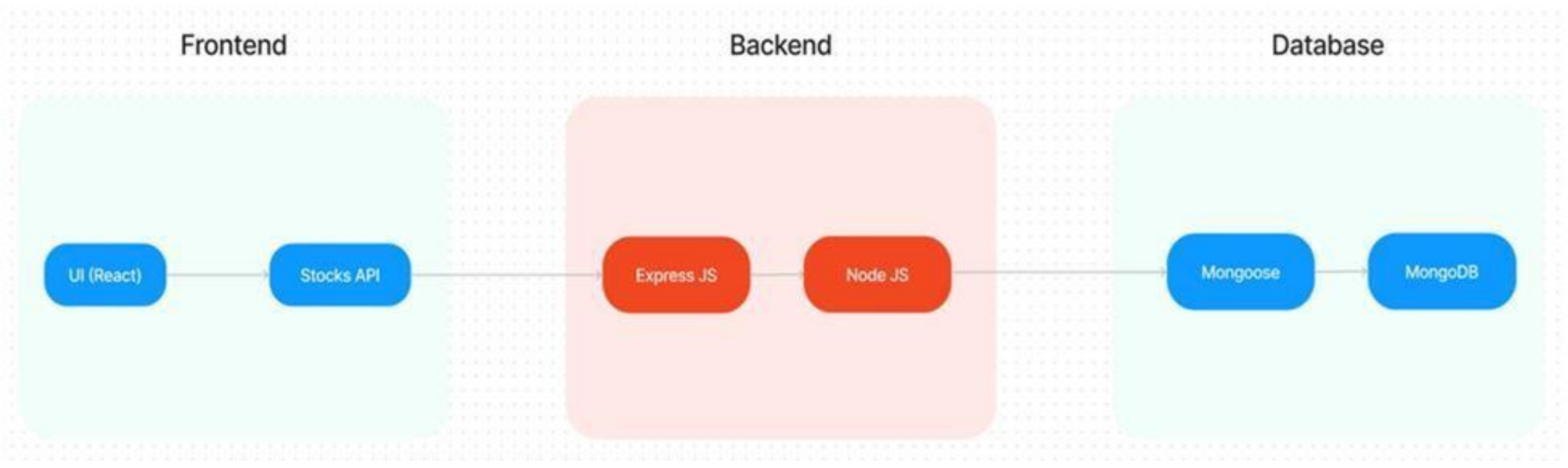
Code Editor: e.g., Visual Studio Code

Web Browsers: Two web browsers installed for testing compatibility (e.g., Chrome and Firefox)

3.3. NETWORK:

Bandwidth: 30 Mbps

4. ARCHITECTURE:



4.1. TECHNICAL ARCHITECTURE:

Designing the technical architecture for a food delivery app, like the hypothetical "SB Food App," involves a combination of frontend, backend, database, and integrations. Below is a high-level overview of the architecture:

1. Architecture Style

- Microservices Architecture: For scalability, maintainability, and modularity.
- API-First Design: RESTful or GraphQL APIs for communication.
- Cloud-Native Deployment: Using cloud platforms like AWS, GCP, or Azure.

2. Core Components

Frontend

- Platform:
  - Mobile: iOS (Swift) and Android (Kotlin) or Cross-platform (Flutter/React Native).
  - Web: React.js or Angular for a Progressive Web App (PWA).
- Features:
  - User onboarding, food catalog, cart management, order tracking, notifications.
- Integrations: Payment gateway SDKs, Google Maps/Apple Maps.

Backend

- Technologies: Node.js, Python (Django/Flask), or Java (Spring Boot).
- Responsibilities:
  - Authentication & Authorization (OAuth2, JWT).

    - Business logic (order processing, delivery tracking).
    - Communication with third-party APIs (payment, maps, notifications).
    - Load balancing and scaling with services like NGINX or AWS Elastic Load Balancer.

Databases

- Primary Database: MongoDB or PostgreSQL for user data, orders, and restaurant details.
- Caching Layer: Redis for fast data retrieval (e.g., frequently accessed menus).
- Search Engine: Elasticsearch for fast and flexible search queries (e.g., nearby restaurants, cuisine).
- Data Warehousing: Snowflake or BigQuery for analytics.

Microservices

- User Management Service: Handles authentication, profile management.
- Restaurant Management Service: Deals with menus, availability, and ratings.
- Order Management Service: Manages cart, order placement, and history.
- Delivery Management Service: Tracks rider location and delivery status.
- Payment Gateway Service: Integrates with Stripe, Razorpay, or PayPal.
- Notification Service: Sends push/email/SMS notifications using Firebase or Twilio.

3. System Design

Scalability

- Auto-scaling via Kubernetes (K8s) or AWS ECS.
- CDN for static content delivery (Cloudflare or AWS CloudFront).

Performance Optimization

- Load Balancer: Distributes traffic across instances.
- CDN for static assets (e.g., images, CSS, JS).
- Asynchronous Processing: Queues with RabbitMQ or Kafka for tasks like order confirmations or delivery updates.

Reliability

- Monitoring: Tools like Prometheus, Grafana, and AWS CloudWatch.
- Error Tracking: Sentry or Datadog.
- Logging: ELK Stack (Elasticsearch, Logstash, Kibana).

Data Security

- Encrypted communication using HTTPS/TLS.
- Secure storage of sensitive data (e.g., hashed passwords with bcrypt).
- Role-based Access Control (RBAC) for admins, users, and delivery agents.

4. Third-Party Integrations

- Payment Gateways: Stripe, Razorpay, or PayPal.

- Maps and Geolocation: Google Maps, Apple Maps, or Mapbox.
- Push Notifications: Firebase Cloud Messaging (FCM), OneSignal.
- Chat Support: Intercom, Twilio, or custom chat using WebSockets.
- Delivery Optimization: APIs like Google Distance Matrix for route optimization.

5. Deployment

- CI/CD Pipeline: GitHub Actions, GitLab CI/CD, or Jenkins.
- Containerization: Docker for service packaging.
- Hosting: AWS, GCP, or Azure with Kubernetes for container orchestration.

6. User Roles

- Customers: Browse, order, pay, track delivery.
- Restaurants: Manage menus, view orders, track revenue.
- Delivery Partners: View delivery requests, update order status.
- Admin: Manage platform settings, analytics, and disputes.

5. ER - DIAGRAM:

Relationships

- User → Order: One user can place multiple orders (1

).
- Order → OrderItem: Each order can have multiple items (1

).
- OrderItem → MenuItem: An order item corresponds to a specific menu item (M:1).
- User → Review: A user can leave multiple reviews (1

).
- Restaurant → Review: Each review belongs to a specific restaurant (M:1).
- Restaurant → Menu: A restaurant can have one or more menus (1

).
- Menu → MenuItem: A menu consists of multiple menu items (1

).
- Order → Payment: Each order has one payment (1:1).
- Order → DeliveryPartner: Each order is delivered by one delivery partner (1:1)

6. SETUP INSTRUCTIONS:

Folder Setup

▶ Create Frontend and Backend folders

A. FRONT-END DEVELOPMENT:

For Frontend, we use require dependencies as follows, they are:

```
Frontend/
 |
├── public/          # Public assets (HTML, images, icons)
|   └── index.html   # Main HTML file

├── src/             # Application source code
|   ├── components/  # Reusable React components
|   ├── pages/    # Page-specific components (e.g., HomePage)
|   ├── assets/      # Images, fonts, stylesheets
|   ├── styles/      # Global and modular styles (CSS/SASS)
|   ├── utils/       # Helper functions and utilities
|   ├── App.js       # Main application component
|   ├── index.js     # Entry point for React
|   └── api/         # API interaction logic
├── .env             # Environment variables (e.g., API URLs)
├── .gitignore       # Ignore unnecessary files for Git
├── package.json     # Project dependencies and scripts
└── README.md        # Documentation
```

B. <u>BACK-END DEVELOPMENT</u>:

Setup express server

- ▶ Create index.js file in the server (backend folder).
- ▶ define port number, mongodb connection string and JWT key in env file to access it.
- ▶ Configure the server by adding cors, body-parser.

```
Backend/

 |

├── src/

|   ├── config/        # Configuration files (e.g., DB connection, environment)

|   ├── contro lers/     # Business logic for API endpoints

|   ├── models/        # Database models/schemas

|   ├── routes/        # API routes

|   ├── middlewares/     # Authentication, validation, etc.

|   ├── services/       # Helper functions (e.g., payment, email, notifications)

|   ├── utils/        # Utility functions

|   ├── app.js         # Main app configuration and middleware

|   └── server.js       # Entry point for the server
```

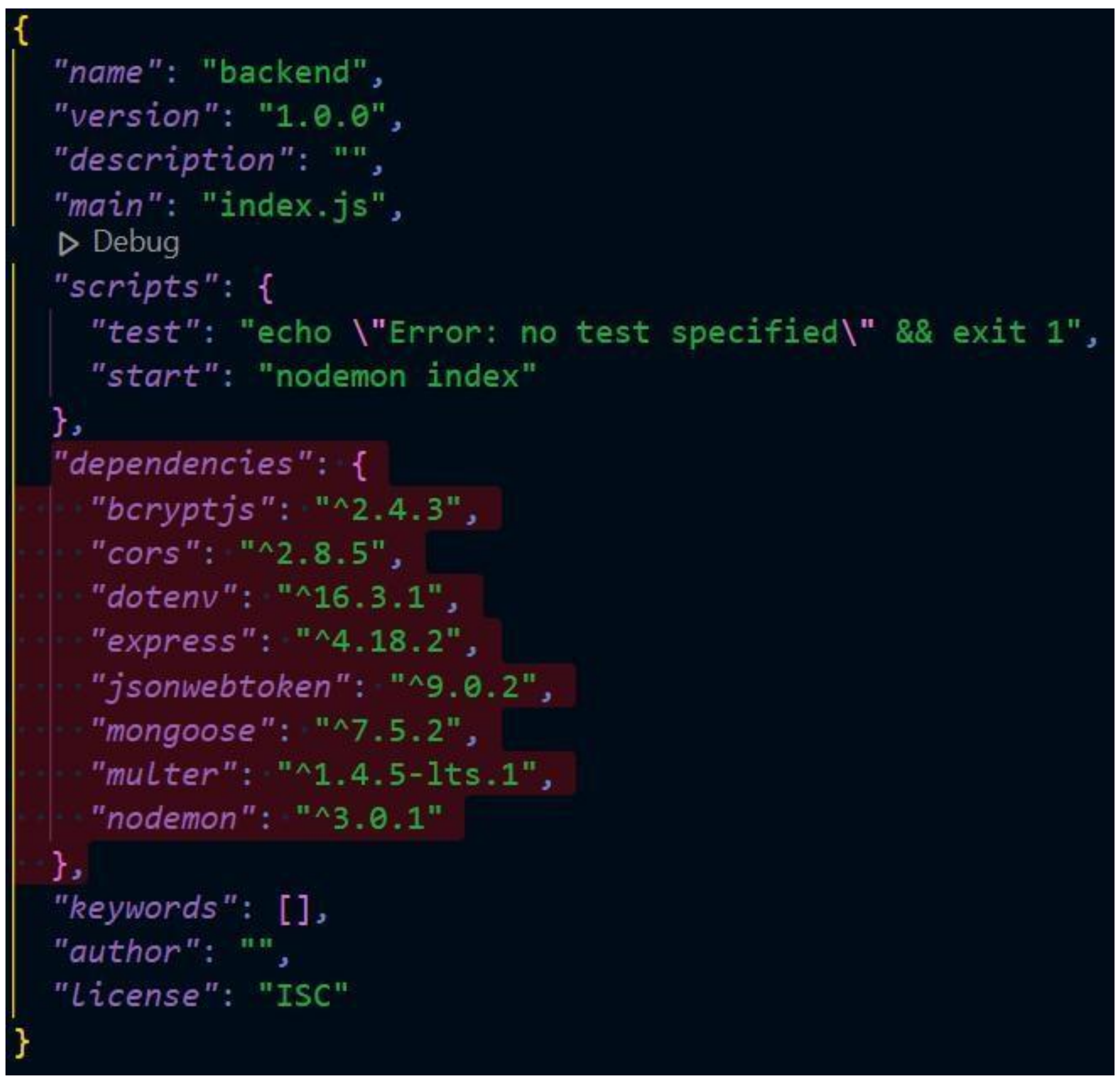
|

├── .env # Environment variables

├── .gitignore # Files to ignore in Git

├── package.json # Dependencies and scripts

└── README.md # Documentation



```
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon index"
  },
  "dependencies": {
    "bcryptjs": "^2.4.3",
    "cors": "^2.8.5",
    "dotenv": "^16.3.1",
    "express": "^4.18.2",
    "jsonwebtoken": "^9.0.2",
    "mongoose": "^7.5.2",
    "multer": "^1.4.5-lts.1",
    "nodemon": "^3.0.1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

C. <u>DATABASE DEVELOPMENT</u>:

Configure MongoDB

▶ Import Mongoose

▶ Add database connection from config.js file present in config folder.

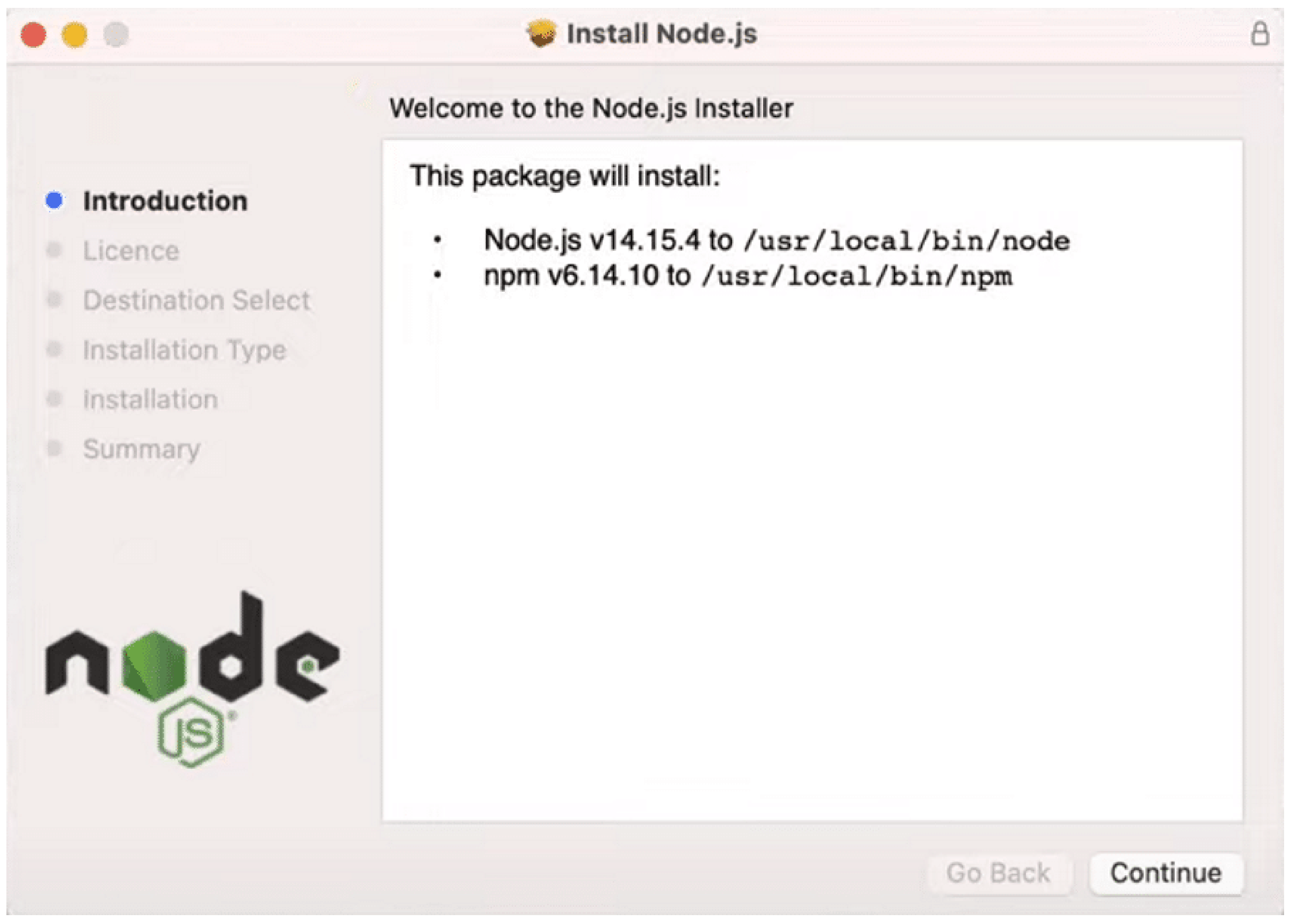
▶ Create a model folder to store all the DB schemas.

6.1. <u>PRE-REQUISITES</u>:

Here are the key prerequisites for developing a fu l-stack application. They are:

1. <u>Development Environment</u>

- Frontend Development:
  - Install Node.js (v16 or later) from [Node.js Official Website](#).
  - Install npm or yarn for package management (comes with Node.js).
  - Install a modern code editor like Visual Studio Code with extensions for JavaScript, React.



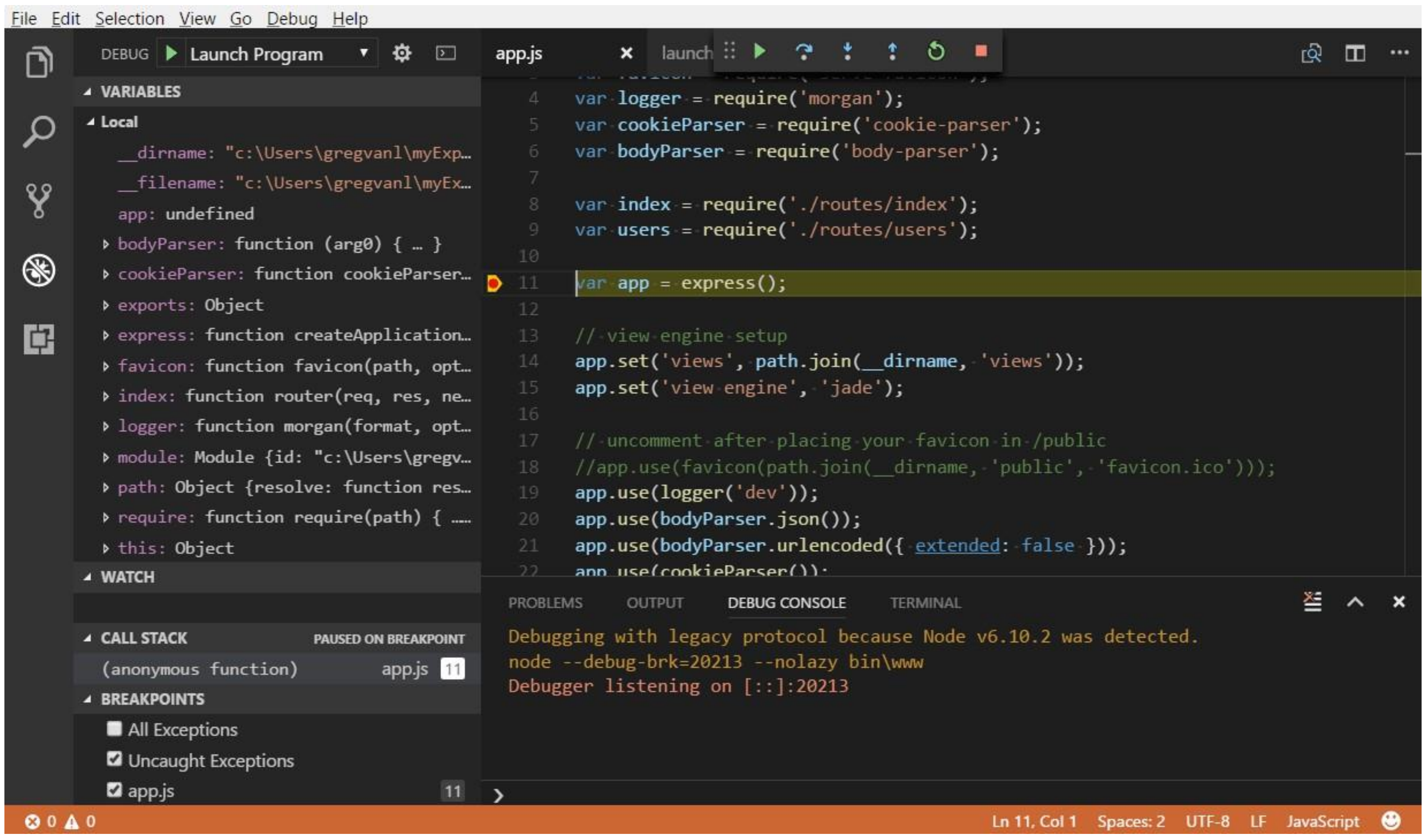
Welcome to the Node.js Installer

This package will install:

- Node.js v14.15.4 to `/usr/local/bin/node`
- npm v6.14.10 to `/usr/local/bin/npm`

Backend Development:

- Install Node.js (v16 or later).
- Install MongoDB (locally or configure MongoDB Atlas for a cloud database).
- Install Postman or URL for testing backend APIs.



```
__dirname: "c:\Users\gregvanl\myExp…
__filename: "c:\Users\gregvanl\myEx…
app: undefined
▸ bodyParser: function (arg0) { … }
▸ cookieParser: function cookieParser…
▸ exports: Object
▸ express: function createApplication…
▸ favicon: function favicon(path, opt…
▸ index: function router(req, res, ne…
▸ logger: function morgan(format, opt…
▸ module: Module {id: "c:\Users\gregv…
▸ path: Object {resolve: function res…
▸ require: function require(path) { ……
▸ this: Object
```

```
(anonymous function)    app.js 11
```

- All Exceptions
- ✓ Uncaught Exceptions
- ✓ app.js 11

```javascript
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var index = require('./routes/index');
var users = require('./routes/users');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
```

```
Debugging with legacy protocol because Node v6.10.2 was detected.
node --debug-brk=20213 --nolazy bin\www
Debugger listening on [::]:20213
```

- Version Control:
  - Install Git and set up a GitHub, GitLab, or Bitbucket repository for version control.

2. Software Dependencies

- Frontend:
  - Install React.js or another frontend framework.
  - Required packages: `react-router-dom`, `axios`, `redux`, `@mui/material`, and more.

    - Optional: SASS for styling, Formik/Yup for form validation.
- Backend:
    - Use Express.js for creating RESTful APIs.
    - Required packages: `mongoose`, `dotenv`, `bcrypt`, `jsonwebtoken`, `cors`, `body-parser`.
    - Optional: `Joi` for validation, `multer` for file uploads.

3. Hardware Requirements

- Developer Machine:
    - Processor: Minimum Intel i5 or equivalent.
    - RAM: At least 8GB (16GB recommended).
    - Disk Space: 10GB free space for software and database.
    - OS: Windows 10/11, macOS, or a Linux-based system.
- Server Requirements (for deployment):
    - Node.js-compatible hosting.
    - At least 1GB RAM and 1 CPU for small-scale testing; scale up based on traffic.
    - Database hosting for MongoDB (e.g., MongoDB Atlas).

4. Accounts and Keys

- API Integrations:
    - Google Maps API for location services.
    - Payment Gateway API keys (e.g., Stripe, PayPal).
    - Firebase (optional) for push notifications or authentication.
- Third-Party Tools:
    - Register accounts for email services like SendGrid or Twilio (SMS notifications).
    - Cloud storage accounts for storing assets (e.g., AWS S3, Google Cloud Storage).

5. Design and Documentation

- Wireframes/Mockups:
    - Use tools like Figma or Adobe XD to create user interface prototypes.
- Database Schema:
    - Define the data structure and relationships before implementation.
- API Documentation:
    - Plan API endpoints using tools like Swagger or Postman co lections.

6. Team and Collaboration Tools

- Team Collaboration:
    - Use Slack or Microsoft Teams for communication.
    - Tre lo, Jira, or Asana for task and project management.
- Version Control and CI/CD:
    - Git for version control.
    - CI/CD pipelines using GitHub Actions, GitLab CI/CD, or Jenkins.

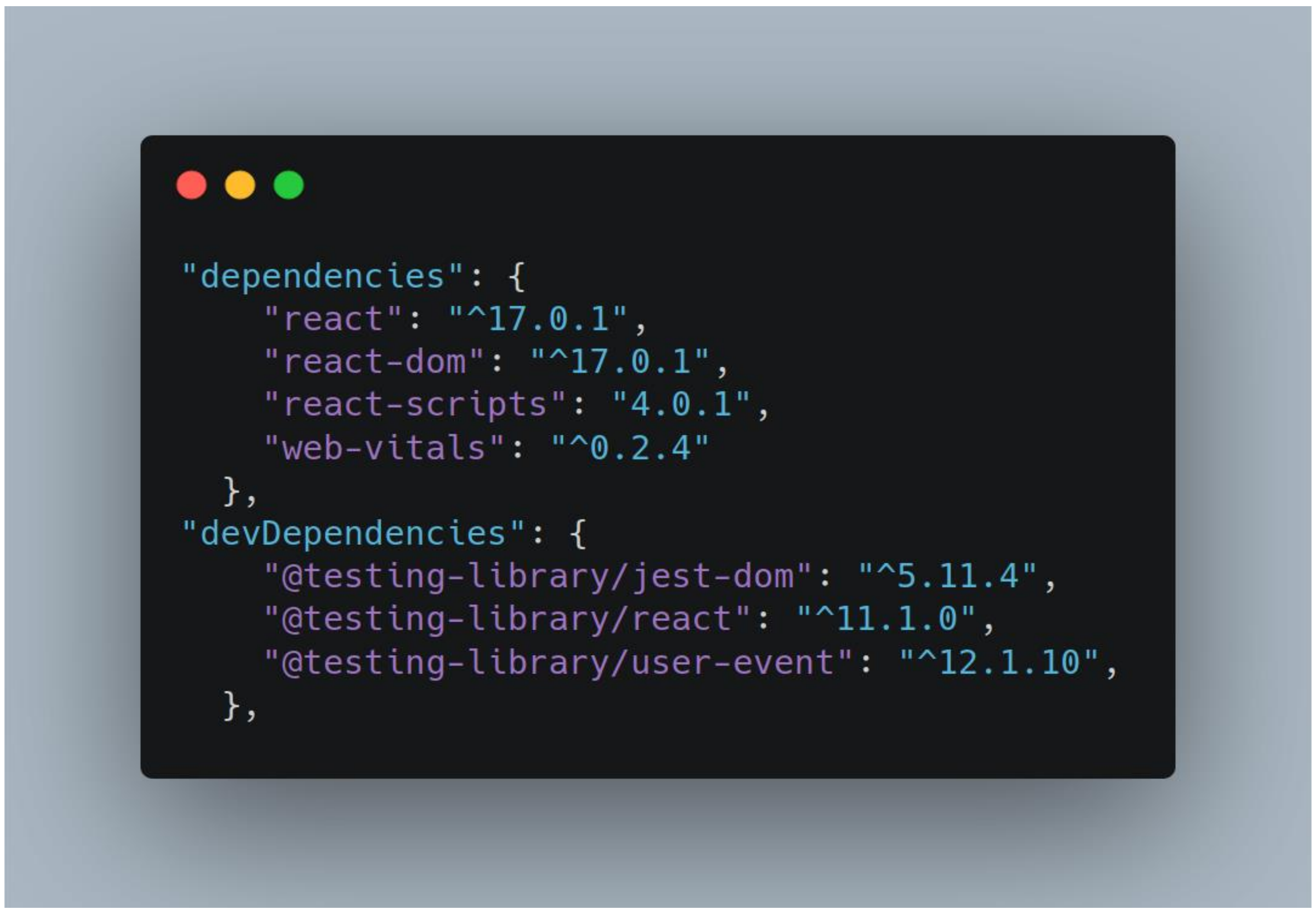
7. Security Measures

- Secure sensitive data with environment variables using **.env** files.
- Implement HTTPS for secure communication.
- Follow OWASP guidelines to prevent common vulnerabilities (e.g., SQL Injection, XSS).

8. Testing Setup

- Install testing libraries:
    - Frontend: Jest, React Testing Library.
    - Backend: Mocha, Chai, Supertest.

React Testing Library.



```
"dependencies": {
    "react": "^17.0.1",
    "react-dom": "^17.0.1",
    "react-scripts": "4.0.1",
    "web-vitals": "^0.2.4"
  },
"devDependencies": {
    "@testing-library/jest-dom": "^5.11.4",
    "@testing-library/react": "^11.1.0",
    "@testing-library/user-event": "^12.1.10",
  },
```

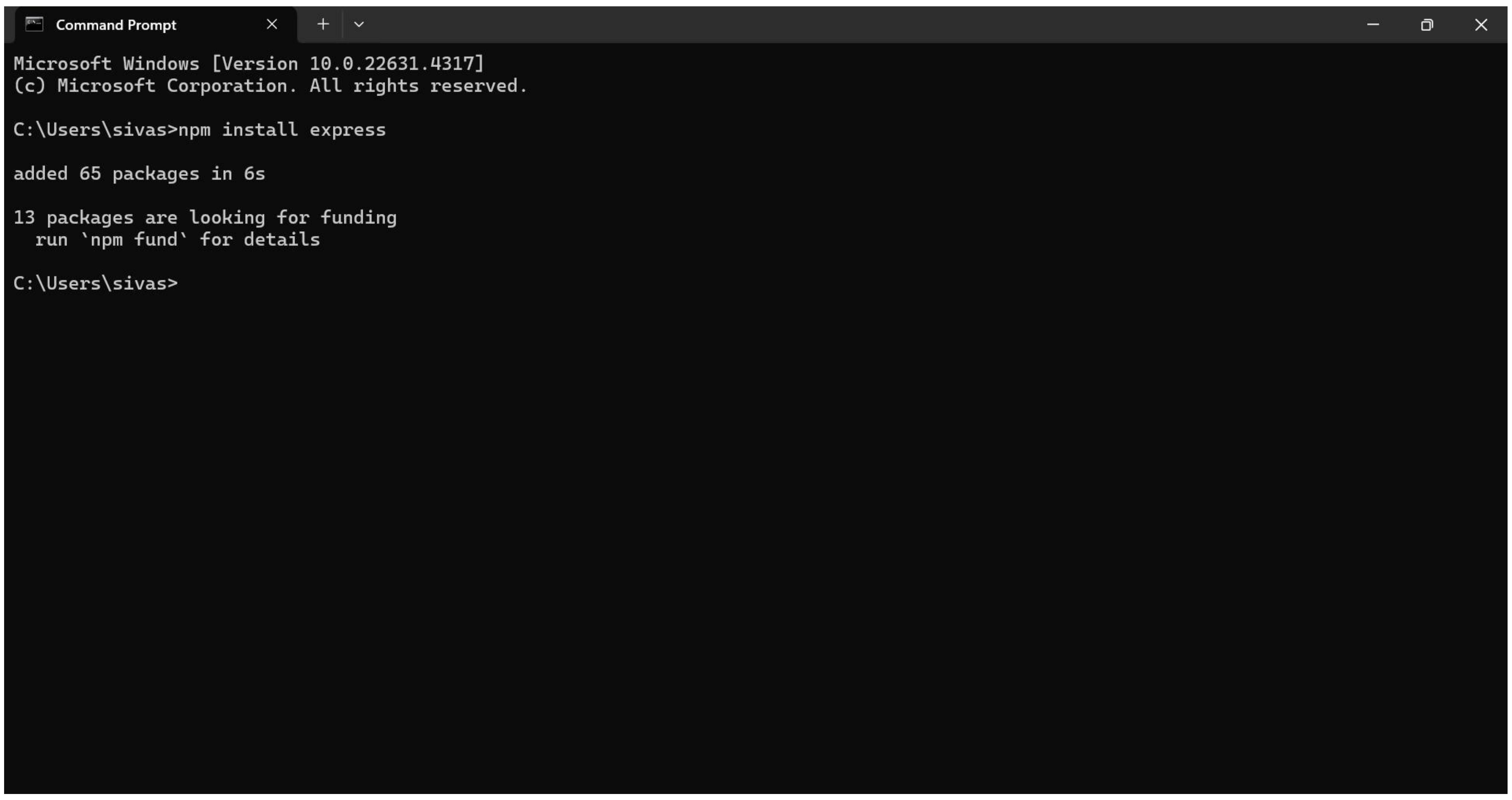
6.2. INSTALLATION:

Installation Steps

1. Clone the Repository

Installation: Open your command prompt or terminal and run the following command:

npm install express



```
Microsoft Windows [Version 10.0.22631.4317]
(c) Microsoft Corporation. All rights reserved.

C:\Users\sivas>npm install express

added 65 packages in 6s

13 packages are looking for funding
  run `npm fund` for details

C:\Users\sivas>
```

✓ MongoDB:

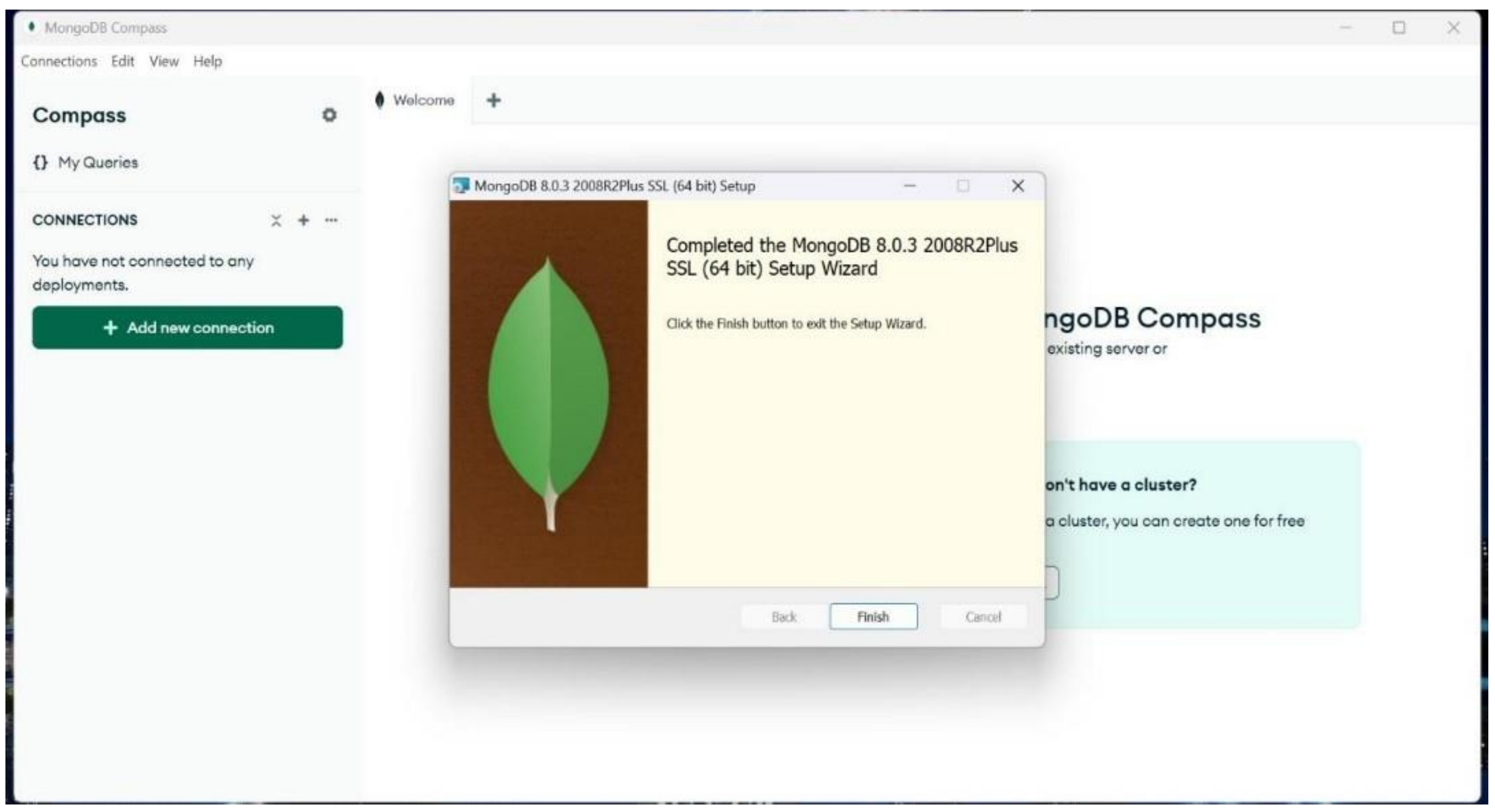
MongoDB is a flexible and scalable NoSQL database that stores data in a JSON-like format. It provides high performance, horizontal scalability, and seamless integration with Node.js, making it ideal for handling large amounts of structured and unstructured data.

Set up a MongoDB database to store your application's data.

Download:

https://www.mongodb.com/try/download/community

Installation instructions:



You have not connected to any deployments.

Completed the MongoDB 8.0.3 2008R2Plus SSL (64 bit) Setup Wizard

Click the Finish button to exit the Setup Wizard.

existing server or

on't have a cluster?

a cluster, you can create one for free

https://docs.mongodb.com/manual/installation/

- √ React.js:

  React.js is a popular JavaScript library for building user interfaces. It enables developers to create interactive and reusable UI components, making it easier to build dynamic and responsive web applications.

  Install React.js, a JavaScript library for building user interfaces.

  Follow the installation guide: https://reactjs.org/docs/create-a-new-react-app.html

- √ HTML, CSS, and JavaScript:

Basic knowledge of HTML for creating the structure of your app, CSS for styling, and JavaScript for client-side interactivity is essential.

- √ Database Connectivity:

  Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose to connect your Node.js server with the MongoDB database and perform CRUD (Create, Read,

  Update, Delete) operations. Before connecting you should set up a environment variable

PROJECT FOLDER STRUCTURE:

1. Root Directory

```
bash
Copy code
SB-Food-App/
│
├── Frontend/        # Frontend application code
├── Backend/         # Backend application code
├── README.md          # Project documentation
├── .gitignore       # Ignored files for Git
└── package.json       # Project-level dependencies (optional)
```

2. Frontend Folder Structure

```
bash
Copy code
Frontend/
│
├── public/            # Static files accessible to the browser
│   └── index.html        # Main HTML file
│
├── src/              # Main source code
│   ├── assets/          # Images, fonts, and other static assets
│   ├── components/       # Reusable UI components
│   ├── pages/           # Page-specific components
│   ├── styles/          # Global and modular styles
│   ├── api/             # API service functions
│   ├── utils/           # Utility/helper functions
│   ├── App.js           # Main app component
│   ├── index.js          # React entry point
│   └── routes.js         # Route definitions for the app
│
├── .env              # Environment variables for the frontend
├── .gitignore           # Ignored files for Git
├── package.json          # Frontend dependencies and scripts
└── README.md            # Frontend-specific documentation
```

Key Directories in Frontend

- public/: Contains static files like index.html, app icons, and metadata.
- components/: Contains reusable UI components such as buttons, modals, and navbars.
- pages/: Includes specific screens or views like HomePage, LoginPage, OrderDetailsPage.
- api/: Centralized API interaction logic using libraries like Axios.
- utils/: Helper functions (e.g., date formatting, currency conversion).

3. Backend Folder Structure

```
bash
Copy code
Backend/
│
├── src/              # Main source code
│   ├── config/          # Configuration files (e.g., DB connection, environment setup)
│   ├── controllers/      # Business logic for routes
│   ├── models/          # MongoDB schemas and data models
│   ├── routes/          # API route handlers
│   ├── middlewares/      # Middleware (e.g., authentication, validation)
│   ├── services/        # External service integrations (e.g., payment, email)
│   ├── utils/          # Utility/helper functions
│   ├── app.js          # Main app configuration
│   └── server.js        # Entry point to start the server
│
├── .env               # Environment variables for the backend
├── .gitignore           # Ignored files for Git
├── package.json         # Backend dependencies and scripts
└── README.md           # Backend-specific documentation
```

Key Directories in Backend

- config/: Configuration for database connections, API keys, etc.
- controllers/: Functions containing the business logic for handling requests.
- models/: Mongoose schemas for defining data structure.
- routes/: Defines all API endpoints and maps them to controllers.
- middlewares/: Functions that execute during request handling (e.g., JWT validation).
- services/: External integrations like payment gateways or notification services.
- utils/: Helper functions like error handling and response formatting.

4. Example Frontend Structure in Detail

```
css
Copy code
Frontend/
│
├── src/
│   ├── assets/
│   │   ├── images/
│   │   │   └── logo.png
│   │   ├── fonts/
│   │   └── styles/
│   │       └── global.css
│   │
│   ├── components/
│   │   ├── Navbar.jsx
│   │   ├── Button.jsx
│   │   └── Footer.jsx
│   │
│   ├── pages/
│   │   ├── HomePage.jsx
│   │   ├── LoginPage.jsx
│   │   └── CartPage.jsx
│   │
│   ├── api/
│   │   ├── axiosInstance.js
│   │   └── userApi.js
│   │
│   ├── utils/
│   │   └── formatCurrency.js
│   │
│   ├── App.js
│   ├── index.js
│   └── routes.js
│
```

5. Example Backend Structure in Detail

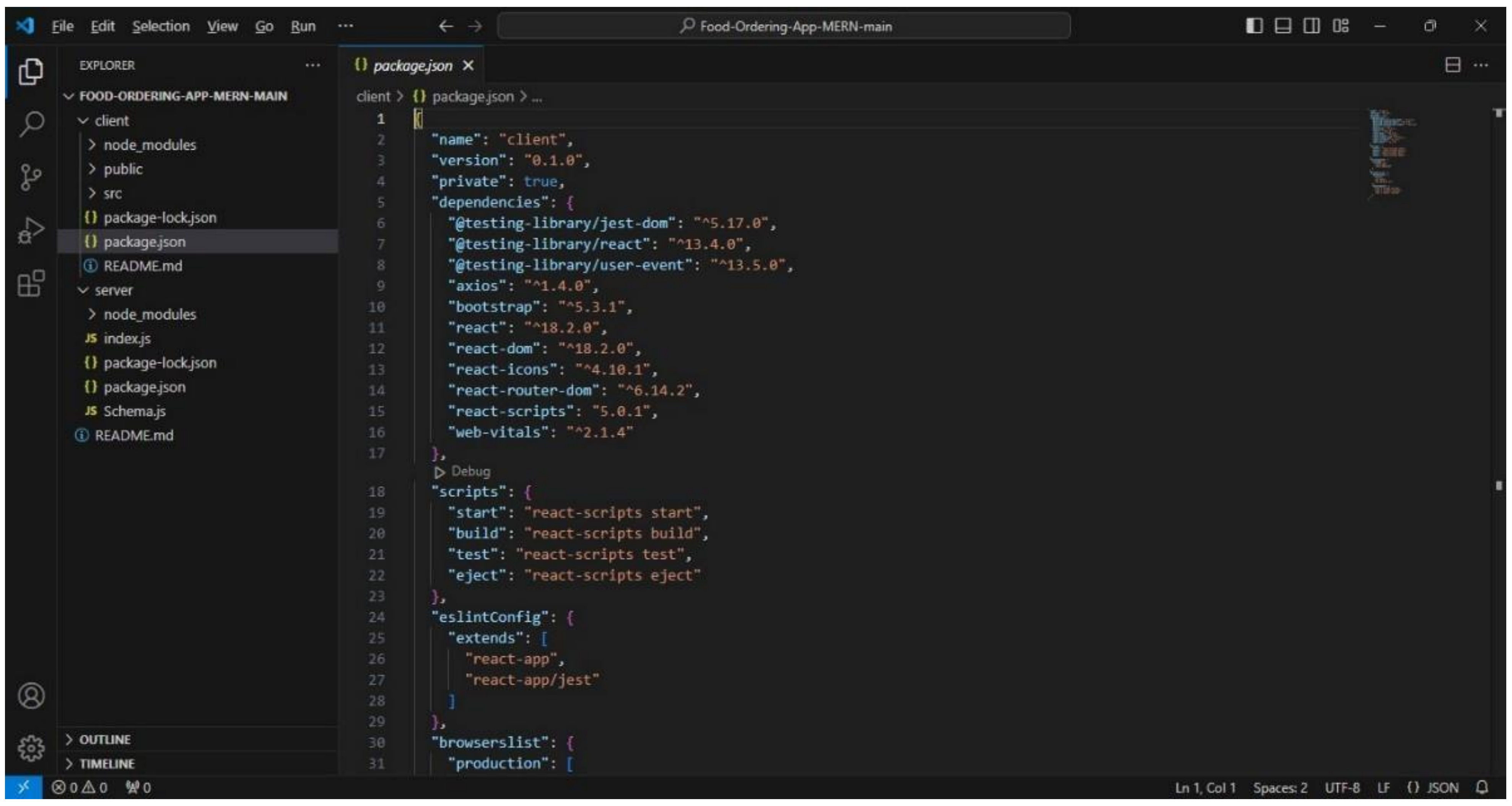
```
css
Copy code
Backend/
│
├── src/
│   ├── config/
│   │   └── db.js       # MongoDB connection logic
│   │
│   ├── controllers/
│   │   ├── userController.js
│   │   └── orderController.js
│   │
│   ├── models/
│   │   ├── User.js
│   │   ├── Order.js
│   │   └── Restaurant.js
│   │
│   ├── routes/
│   │   ├── userRoutes.js
│   │   └── orderRoutes.js
│   │
│   ├── middlewares/
│   │   ├── authMiddleware.js
│   │   └── errorHandler.js
│   │
│   ├── services/
│   │   ├── paymentService.js
│   │   └── emailService.js
│   │
│   ├── utils/
│   │   └── logger.js
│   │
│   ├── app.js
│   └── server.js
│
```

6. Combined Folder Structure Overview

```
java
Copy code
SB-Food-App/
│
├── Frontend/
│   ├── public/
│   ├── src/
│   │   ├── components/
│   │   ├── pages/
│   │   ├── api/
│   │   ├── styles/
│   │   ├── App.js
│   │   └── index.js
│   ├── .env
│   └── package.json
│
├── Backend/
│   ├── src/
│   │   ├── config/
│   │   ├── controllers/
│   │   ├── models/
│   │   ├── routes/
│   │   ├── middlewares/
│   │   │
│    │    ├── services/
│    │    ├── utils/
│    │    ├── app.js
│    │    └── server.js
│    ├── .env
│    └── package.json
│
├── README.md
├── .gitignore
└── package.json
```

README.md

- A markdown file that typically contains documentation for the project. It often includes an introduction, setup instructions, usage information, and any other details that help developers understand and contribute to the project.



```
FOOD-ORDERING-APP-MERN-MAIN
  client
    node_modules
    public
    src
    package-lock.json
    package.json
    README.md
  server
    node_modules
    index.js
    package-lock.json
    package.json
    Schema.js
  README.md
```

```json
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.17.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "axios": "^1.4.0",
    "bootstrap": "^5.3.1",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-icons": "^4.10.1",
    "react-router-dom": "^6.14.2",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
```

7. RUNNING THE APPLICATION:

To run the Online Learning Platform (OLP) application locally, follow these steps:

For Backend:

- ▶ Open Terminal in VS Code or Command Prompt
- ▶ Navigate to the backend directory with the command: cd backend
- ▶ Install dependencies if not installed already: npm install
- ▶ Start the backend server: npm start
- ▶ The backend will run on http:/ localhost:8000 in web browser to test API routes

For Frontend:

- ▶ Open Terminal in VS Code or Command Prompt
- ▶ Navigate to the frontend directory with the command: cd frontend
- ▶ Install dependencies if not installed already: npm install

- Start the frontend server: npm run dev
- The frontend will run on http://localhost:5173 in web browser to access application

10. API DOCUMENTATION:

The following is a documentation of the API endpoints exposed by the backend of the OLP. These endpoints handle functionalities such as user registration, login, course management, enrollment, and more.

BACKEND CODE SNIPPETS WITH API

Backend > config > index.js

```
import express from 'express'
import bodyParser from 'body-parser';
import mongoose from 'mongoose';
import cors from 'cors';
import bcrypt from 'bcrypt';
import {Admin, Cart, FoodItem, Orders, Restaurant, User } from './Schema.js'


const app = express();

app.use(express.json());
app.use(bodyParser.json({limit: "30mb", extended: true}))
app.use(bodyParser.urlencoded({limit: "30mb", extended: true}));
app.use(cors());

const PORT = 6001;

mongoose.connect('mongodb://localhost:27017/foodDelivery',{
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(()=>{

  app.post('/register', async (req, res) => {
    const { username, email, usertype, password ,  restaurantAddress, restaurantImage}  =  req.body;
    try {

      const  existingUser  =  await User.findOne({  email  });
      if (existingUser) {
        return res.status(400).json({ message: 'User already exists' });
      }

      const hashedPassword = await bcrypt.hash(password, 10);

      if(usertype === 'restaurant'){
        const newUser = new User({
          username, email, usertype, password: hashedPassword, approval: 'pending'
        });
        const user =  await newUser.save();
        console.log(user._id);
```

```
            const restaurant = new Restaurant({ownerId: user._id ,title: username, address: restaurantAddress,
mainImg: restaurantImage, menu: []});
            await restaurant.save();

            return res.status(201).json(user);

        } else{

            const newUser = new User({
               username, email, usertype, password: hashedPassword, approval: 'approved'
            });
            const userCreated = await newUser.save();
            return res.status(201).json(userCreated);
        }


    } catch (error) {
      console.log(error);
      return res.status(500).json({ message: 'Server Er or' });
    }
  });



  app.post('/login', async (req, res) => {
    const { email, password } = req.body;
    try {

        const user = await User.findOne({ email });

        if (!user) {
            return res.status(401).json({ message: 'Invalid email or password' });
        }
        const isMatch = await bcrypt.compare(password, user.password);
        if (!isMatch) {
            return res.status(401).json({ message: 'Invalid email or password' });
        } else{
            return res.json(user);
        }

    } catch (error) {
      console.log(error);
      return res.status(500).json({ message: 'Server Er or' });
    }
  });

  // promote restaurant

  app.post('/update-promote-list', async(req, res)=>{
    const {promoteList} = req.body;
    try{
        const admin = await Admin.findOne();
        admin.promotedRestaurants = promoteList;
        await admin.save();
        res.json({message: 'approved'});

    }catch(err){
        res.status(500).json({ message: 'Error occured' });
    }
  })

  // approve restaurant

  app.post('/approve-user', async(req, res)=>{
    const {id} = req.body;
    try{
        const restaurant = await User.findById(id);
        restaurant.approval = 'approved';
        await restaurant.save();
        res.json({message: 'approved'});

    }catch(err){
        res.status(500).json({ message: 'Error occured' });
    }
})

// reject restaurant

app.post('/reject-user', async(req, res)=>{
    const {id} = req.body;
    try{
        const restaurant = await User.findById(id);
        restaurant.approval = 'rejected';
        await restaurant.save();
        res.json({message: 'rejected'});

    }catch(err){
        res.status(500).json({ message: 'Error occured' });
    }
})



// fetch user details

app.get('/fetch-user-details/:id', async(req, res)=>{

    try{
        const user = await User.findById(req.params.id);
        res.json(user);

    }catch(err){
        res.status(500).json({ message: 'Error occured' });
    }
})

// fetch users

app.get('/fetch-users', async(req, res)=>{
    try{
        const users = await User.find();
        res.json(users);

    }catch(err){
        res.status(500).json({ message: 'Error occured' });
    }
})

// fetch restaurants

app.get('/fetch-restaurants', async(req, res)=>{
    try{
        const restaurants = await Restaurant.find();
        res.json(restaurants);

    }catch(err){
        res.status(500).json({ message: 'Error occured' });
    }
})
```

```
//  // Fetch individual product
//  app.get('/fetch-product-details/:id', async(req, res)=>{
//     const id = req.params.id;
//     try{
//        const product = await Product.findById(id);
//        res.json(product);
//     }catch(err){
//        res.status(500).json({message: "Er or occured"});
//     }
// })

// // fetch products

// app.get('/fetch-products', async(req, res)=>{
//     try{
//        const products = await Product.find();
//        res.json(products);

//     }catch(err){
//        res.status(500).json({ message: 'Er or occured' });
//     }
// })

// fetch orders

app.get('/fetch-orders', async(req, res)=>{
   try{
      const orders = await Orders.find();
      res.json(orders);

   }catch(err){
      res.status(500).json({ message: 'Error occured' });
   }
})

// fetch food items

app.get('/fetch-items', async(req, res)=>{
   try{
      const items = await FoodItem.find();
      res.json(items);

   }catch(err){
      res.status(500).json({ message: 'Error occured' });
   }
})


// Fetch categories

app.get('/fetch-categories', async(req, res)=>{
   try{
      const data = await Admin.find();
      if(data.length===0){
         const newData = new Admin({ categories: [], promotedRestaurants: []})
         await newData.save();
         return res.json(newData[0].categories);
      }else{
         return res.json(data[0].categories);
      }
   }catch(err){
      res.status(500).json({message: "Error occured"});
   }
})
```

```
// Fetch promoted list

app.get('/fetch-promoted-list', async(req, res)=>{
  try{
    const data = await Admin.find();
    if(data.length===0){
      const newData = new Admin({ categories: [], promotedRestaurants: []})
      await newData.save();
      return res.json(newData[0].promotedRestaurants);
    }else{
      return res.json(data[0].promotedRestaurants);
    }
  }catch(err){
    res.status(500).json({message: "Error occured"});
  }
})

// fetch restaurant details with owner id

app.get('/fetch-restaurant-details/:id', async(req, res)=>{

  try{
    const restaurant = await Restaurant.findOne({ownerId: req.params.id});
    res.json(restaurant);

  }catch(err){
    res.status(500).json({ message: 'Error occured' });
  }
})



 // fetch restaurant details with restaurant id

 app.get('/fetch-restaurant/:id', async(req, res)=>{

  try{
    const restaurant = await Restaurant.findById( req.params.id);
    res.json(restaurant);

  }catch(err){
    res.status(500).json({ message: 'Error occured' });
  }
})

 // fetch item details

 app.get('/fetch-item-details/:id', async(req, res)=>{

  try{
    const item = await FoodItem.findById(req.params.id);
    res.json(item);

  }catch(err){
    res.status(500).json({ message: 'Error occured' });
  }
})



// Add new product

app.post('/add-new-product', async(req, res)=>{
  const {restaurantId, productName, productDescription, productMainImg, productCategory,
productMenuCategory, productNewCategory, productPrice, productDiscount} = req.body;
```

```
        try{
          if(productMenuCategory === 'new category'){
            const admin = await Admin.findOne();
            admin.categories.push(productNewCategory);
            await admin.save();
            const newProduct = new FoodItem({restaurantId, title: productName, description:
productDescription, itemImg: productMainImg, category: productCategory, menuCategory:
productNewCategory, price: productPrice, discount: productDiscount, rating: 0});
            await newProduct.save();
            const restaurant = await Restaurant.findById(restaurantId);
            restaurant.menu.push(productNewCategory);
            await restaurant.save();
          } else{
            const newProduct = new FoodItem({restaurantId, title: productName, description:
productDescription, itemImg: productMainImg, category: productCategory, menuCategory:
productMenuCategory,  price: productPrice, discount: productDiscount, rating: 0});
            await newProduct.save();
          }
          res.json({message: "product added!!"});
        }catch(err){
          res.status(500).json({message: "Error occured"});
        }
    })




    // update product

    app.put('/update-product/:id', async(req, res)=>{
        const {restaurantId, productName, productDescription, productMainImg, productCategory,
productMenuCategory, productNewCategory, productPrice, productDiscount} = req.body;
        try{
          if(productCategory === 'new category'){
            const admin = await Admin.findOne();
            admin.categories.push(productNewCategory);
            await admin.save();

            const product = await FoodItem.findById(req.params.id);

            product.title = productName;
            product.description = productDescription;
            product.itemImg = productMainImg;
            product.category = productCategory;
            product.menuCategory  =  productNewCategory
            product.price = productPrice;
            product.discount = productDiscount;

            await product.save();

          } else{
            const product = await FoodItem.findById(req.params.id);

            product.title = productName;
            product.description = productDescription;
            product.itemImg = productMainImg;
            product.category = productCategory;
            product.menuCategory  =  productMenuCategory;
            product.price = productPrice;
            product.discount = productDiscount;

            await product.save();
          }
          res.json({message: "product updated !"});
        }catch(err){
            res.status(500).json({message: "Error occured"});
        }
    })


    // cancel order

    app.put('/cancel-order', async(req, res)=>{
        const {id} = req.body;
        try{

            const order = await Orders.findById(id);
            order.orderStatus = 'cancelled';
            await  order.save();
            res.json({message: 'order cancelled'});

        }catch(err){
            res.status(500).json({message: "Error occured"});
        }
    })


    // update order status

    app.put('/update-order-status', async(req, res)=>{
        const {id, updateStatus} = req.body;
        try{

            const order = await Orders.findById(id);
            order.orderStatus = updateStatus;
            await order.save();
            res.json({message: 'order status updated'});

        }catch(err){
            res.status(500).json({message: "Error occured"});
        }
    })


    // fetch cart items

    app.get('/fetch-cart', async(req, res)=>{
        try{

            const items = await Cart.find();
            res.json(items);

        }catch(err){
            res.status(500).json({message: "Error occured"});
        }
    })


    // add cart item

    app.post('/add-to-cart', async(req, res)=>{

        const {userId, foodItemId, foodItemName, restaurantId, foodItemImg, price, discount, quantity} =
req.body
        try{

            const restaurant = await Restaurant.findById(restaurantId);

            const item = new Cart({userId, foodItemId, foodItemName, restaurantId, restaurantName: restaurant.
title, foodItemImg, price, discount, quantity});
```

```
        await item.save();

        res.json({message: 'Added to cart'});

    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
  })




  // remove from cart

  app.put('/remove-item', async(req, res)=>{
    const {id} = req.body;
    try{
        const item = await Cart.deleteOne({_id: id});
        res.json({message: 'item removed'});
    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
  });


  // Order from cart

  app.post('/place-cart-order', async(req, res)=>{
    const {userId, name, mobile, email, address, pincode, paymentMethod, orderDate} = req.body;
    try{

        const cartItems = await Cart.find({userId});
        cartItems.map(async (item)=>{

            const newOrder = new Orders({userId, name, email, mobile, address, pincode, paymentMethod,
orderDate, restaurantId: item.restaurantId, restaurantName: item.restaurantName, foodItemId:
item.foodItemId, foodItemName: item.foodItemName, foodItemImg: item.foodItemImg, quantity:
item.quantity, price: item.price, discount: item.discount})
            await newOrder.save();
            await Cart.deleteOne({_id: item._id})
        })
        res.json({message: 'Order placed'});

    }catch(err){
        res.status(500).json({message: "Error occured"});
    }
  })



  app.listen(PORT, ()=>{
    console.log('running @ 6001');
  })
}).catch((e)=> console.log(`Error in db connection ${e}`));
```

Frontend > index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <!--
      manifest.json provides metadata used when your web app is installed on
a
      user's mobile device or desktop. See
https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.
css" rel="stylesheet"
integrity="sha384-EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd
3yD65VohhpuuCOmLASjC" crossorigin="anonymous">
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

      You can add webfonts, meta tags, or analytics to this file.
      The build step will place the bundled scripts into the <body> tag.

      To begin the development, run `npm start` or `yarn start`.
      To create a production bundle, use `npm run build` or `yarn build`.
    -->
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.
min.js"
integrity="sha384-MrcW6ZMFYlzcLA8Nl+NtUVF0sA7MsXsP1UyJoMp4YLE
uNSfAP+JcXn/tWtIaxVXM" crossorigin="anonymous"></script>
  </body>
</html>
```

```
const handleLogout = () => {
  localStorage.removeItem("token");
  localStorage.removeItem("user");
  window.location.href = "/";
}
const handleOptionClick = (component) => {
  setSelectedComponent(component);
};
return
(
      <Navbar expand="lg" className="bg-body-tertiary">
        <Container fluid>
          <Navbar.Brand>
            <h3>Study App</h3>
          </Navbar.Brand>
          <Navbar.Toggle aria-controls="navbarScroll" />
          <Navbar.Co lapse id="navbarScroll">
    <Nav className="me-auto my-2 my-lg-0" style={{ maxHeight: '100px' }}
    navbarScroll>
              <NavLink  onClick={()  =>  handleOptionClick('home')}>Home</NavLink>
              {user.userData.type === 'Teacher' && (
                <NavLink onClick={() => handleOptionClick('addcourse')}>Add
                Course</NavLink>
              )}
              {user.userData.type === 'Admin' && (
                <>
                  <NavLink onClick={() =>
                handleOptionClick('cousres')}>Courses</NavLink>   </>
              )}
              {user.userData.type === 'Student' && (
                <>
                  <NavLink onClick={() => handleOptionClick('enrolledcourese')}>Enrolled
  Courses</NavLink>
                </>
              )}
            </Nav>
            <Nav>
              <h5 className='mx-3'>Hi {user.userData.name}</h5>
              <Button onClick={handleLogout} size='sm' variant='outline-danger'>Log
            Out</Button > </Nav>
          </Navbar.Co lapse>
        </Container>
      </Navbar>
    )
  } export default

  NavBar;
```

Frontend > src > Components > common > Login.jsx

```
import React, { useState } from 'react' import {
Link, useNavigate } from 'react-router-dom';
import Navbar from 'react-bootstrap/Navbar';
import { Container, Nav } from
'react-bootstrap'; import Avatar from
'@mui/material/Avatar'; import Button from
'@mui/material/Button'; import TextField from
'@mui/material/TextField'; import Grid from
'@mui/material/Grid'; import Box from
'@mui/material/Box';
```

```
import Typography from
'@mui/material/Typography'; import axiosInstance
from './AxiosInstance';

const Login = () => {
  const navigate = useNavigate()
  const [data, setData] =
  useState({
    email: "",
    password: "",
  })
  const handleChange = (e) => {
    const { name, value } = e.target;
    setData({ ...data, [name]: value }); };
  const handleSubmit = (e) => {
    e.preventDefault();
 if (!data?.email || !data?.password) {
      return alert("Please fill all fields");
    } else {
      axiosInstance.post('/api/user/login', data)
        .then((res) => { if
          (res.data.success)
          {
            alert(res.data.message)

            localStorage.setItem("token", res.data.token);
            localStorage.setItem("user",
            JSON.stringify(res.data.userData));
            navigate('/dashboard') setTimeout(() => {
              window.location.reload()
            }, 1000)
          } else {
            alert(res.data.message)
          }
        })
        .catch((err) => { if (err.response &&
          err.response.status === 401) {
            alert("User doesn't exist");
          }
          navigate("/login");
        });
}
  };
  return
  (
    <>
      <Navbar expand="lg" className="bg-body-tertiary">
        <Container fluid>
          <Navbar.Brand><h2>Study App</h2></Navbar.Brand>
          <Navbar.Toggle aria-controls="navbarScroll" />
          <Navbar.Co lapse id="navbarScroll">
```

```
            <Nav  className="me-auto
              my-2 my-lg-0" style={{
              maxHeight: '100px' }}
              navbarScroll
            >
            </Nav>
            <Nav>
              <Link to={'/'}>Home</Link>
              {/* <Link to={'/about'}>About</Link> */}
              <Link to={'/login'}>Login</Link>
              <Link to={'/register'}>Register</Link>
            </Nav>

          </Navbar.Co lapse>
        </Container>
      </Navbar>
      <div className='first-container'>
        <Container component="main" style={{ display: 'flex', justifyContent: 'center',
alignItems: 'center' }} >
          <Box
            sx={{
              marginTop: 8,
              marginBottom: 4,
              display: 'flex',
              flexDirection: 'column',
              alignItems: 'center',
              padding: '10px',
              background:
              '#dddde8db', border:
              '1px solid lightblue',
              borderRadius: '5px'
            }}
          >
            <Avatar sx={{ bgcolor: 'secondary.main' }}>
            </Avatar>
            <Typography component="h1"
              variant="h5"> Sign In
            </Typography>
            <Box component="form" onSubmit={handleSubmit} noValidate>
              <TextField
                margin="normal"
                fullWidth id="email"
                label="Email
                Address"
                name="email"
                value={data.email}
                 onChange={handleChange}
                 autoComplete="email"
                 autoFocus
              />
              <TextField
                 margin="normal"
                 fullWidth
                 name="password"
                 value={data.password}

              onChange={handleChang
                 e} label="Password"
                 type="password"
                 id="password"
                 autoComplete="current-password"
              />
              <Box mt={2}>
                 <Button
                   type="submit"
                   variant="containe
                   d"
                   sx={{ mt: 3, mb: 2 }}
                   style={{ width: '200px' }}
                 >
                   Sign In
                 </Button>
              </Box>
              <Grid container>
                 <Grid item>Have an account?
                   <Link style={{ color: "blue" }} to={'/register'}
                     variant="body2"> {" Sign Up"}
                   </Link>
                 </Grid>
              </Grid>
            </Box>
          </Box>
        </Container>
      </div>
    </>
  )
} export default

Login
```

Frontend > src > Components > common > Footer.jsx

```
import React from 'react'
import '../styles/Footer.css'

const Footer = () => {
  return (
    <div className="Footer">
      <h4>@SB Foods - Have a
feast with the tasty food everyday.
...</h4>
```

```
     <div
className="footer-body">

       <ul>
         <li>Biriyani</li>
         <li>Pizza</li>
       </ul>

       <ul>
         <li>Beverages</li>
         <li>Burger</li>
       </ul>

       <ul>
         <li>Pulav's</li>
         <li>Rice bowls</li>
       </ul>

       <ul>
         <li>Fried Momo's</li>
         <li>Chicken</li>
       </ul>

       <ul>
         <li>Sandwich</li>
         <li>BBQ</li>
       </ul>


     </div>
     <div
className="footer-bottom">
       <p>@ sb-foods.com - All
rights reserved</p>
     </div>
   </div>
  )
}

export default Footer
```

Frontend > src > Components > common > Login.jsx

```
import React, { useContext, useState } from 'react'
import { GeneralContext } from '../context/GeneralContext';

const Login = ({setIsLogin}) => {

  const {setEmail, setPassword, login} = useContext(GeneralContext);
  const handleLogin = async (e) =>{
    e.preventDefault();
    await login();
  }
  return (
    <form className="authForm">
        <h2>Login</h2>
        <div className="form-floating mb-3 authFormInputs">
            <input type="email" className="form-control" id="floatingInput"
placeholder="name@example.com"
                                                        onChange={(e) => setEmail(e.target.value)} />
            <label htmlFor="floatingInput">Email address</label>
        </div>
            <div className="form-floating mb-3 authFormInputs">
            <input type="password" className="form-control" id="floatingPassword"
placeholder="Password"
                                                        onChange={(e) => setPassword(e.target.value)} />
            <label htmlFor="floatingPassword">Password</label>
        </div>
        <button type="submit" className="btn btn-primary" onClick={handleLogin}>Sign
in</button>

        <p>Not registered? <span onClick={()=> setIsLogin(false)}>Register</span></p>
    </form>
  )
}
export default Login
```

Frontend > src > Components > common > Navbar.jsx

```
import React, { useContext, useEffect, useState } from 'react'
import {BsCart3, BsPersonCircle} from 'react-icons/bs'
import {FcSearch} from 'react-icons/fc'
import '../styles/Navbar.css'
import { useNavigate } from 'react-router-dom'
import { GeneralContext } from '../context/GeneralContext'
import axios from 'axios'
import {ImCancelCircle} from 'react-icons/im'

const Navbar = () => {


  const navigate = useNavigate();

  const usertype = localStorage.getItem('userType');
  const username = localStorage.getItem('username');

  const {logout, cartCount} = useContext(GeneralContext);
  const [productSearch, setProductSearch] = useState('');

  const [noResult, setNoResult] = useState(false);
  const [categories, setCategories] = useState([]);

  useEffect(()=>{
    fetchData();
  }, [])

  const fetchData = async() =>{

    await axios.get('http://localhost:6001/fetch-categories').then(
      (response)=>{
        setCategories(response.data);
      }
    )
  }


  const handleSearch = () =>{
    if (categories.includes(productSearch)){
      navigate(`/category/${productSearch}`);
    }else{
      setNoResult(true);
    }
  }

  return (

    <>
      {/* user navbar */}

        {!usertype ?

            <div className="navbar">
              <h3 onClick={()=> navigate('')}>SB Foods</h3>
              <div className="nav-content">
                <div className="nav-search">
                  <input type="text" name="nav-search" id="nav-search" placeholder='Search
  Restaurants, cuisine, etc.,' onChange={(e)=>setProductSearch(e.target.value)} />
                  <FcSearch className="nav-search-icon" onClick={handleSearch} />
                    {
                      noResult === true ?
                        <div className='search-result-data'>no items found.... try searching for
  Biriyani, Pizza, etc., <ImCancelCircle className='search-result-data-close-btn'
onClick={()=> setNoResult(false)} /></div>
                      :
                      ""

                    }
                </div>
                <button className='btn btn-outline-primary' onClick={()=>
navigate('/auth')}>Login</button>

           </div>
         </div>

       :
       <>
         {usertype === 'customer' ?
              <div className="navbar">
                <h3 onClick={()=> navigate('')}>SB Foods</h3>
                <div className="nav-content">
                  <div className="nav-search">
                    <input type="text" name="nav-search" id="nav-search"
placeholder='Search Restaurants, cuisine, etc.,'
onChange={(e)=>setProductSearch(e.target.value)}  />
                    <FcSearch className="nav-search-icon" onClick={handleSearch} />
                    {
                      noResult === true ?
                        <div className='search-result-data'>no items found.... try searching for
Biriyani, Pizza, etc., <ImCancelCircle className='search-result-data-close-btn'
onClick={()=> setNoResult(false)}  /></div>
                      :
                      ""
                    }
                  </div>

                  <div   className='nav-content-icons'   >
                    <div className="nav-profile" onClick={()=> navigate('/profile')}>
                      <BsPersonCircle className='navbar-icons' data-bs-toggle="tooltip"
data-bs-placement="bottom" title="Profile" />
                      <p>{username}</p>
                    </div>
                    <div  className="nav-cart"  onClick={()=>  navigate('/cart')}>
                      <BsCart3 className='navbar-icons' data-bs-toggle="tooltip"
data-bs-placement="bottom" title="Cart" />
                      <div className="cart-count">{cartCount}</div>
                    </div>
                  </div>

                </div>
              </div>
         :
         <>
            {usertype === 'admin' ?
              <div className="navbar-admin">
                <h3 onClick={()=> navigate('/admin')}>SB Foods (admin)</h3>

                <ul>
                  <li onClick={()=> navigate('/admin')}>Home</li>
                  <li onClick={()=> navigate('/all-users')}>Users</li>
                  <li onClick={()=> navigate('/all-orders')}>Orders</li>
                  <li onClick={()=> navigate('/all-restaurants')}>Restaurants</li>
                  <li onClick={()=> logout()}>Logout</li>
                </ul>
              </div>

              :

              <>
               {usertype === 'restaurant' ?
                  <div className="navbar-admin">
                    <h3 onClick={()=> navigate('/restaurant')}>SB Foods (Restaurant)</h3>

                    <ul>
                      <li onClick={()=> navigate('/restaurant')}>Home</li>
                      <li onClick={()=> navigate('/restaurant-orders')}>Orders</li>
                      <li onClick={()=> navigate('/restaurant-menu')}>Menu</li>
                      <li onClick={()=> navigate('/new-product')}>New Item</li>
                      <li onClick={()=> logout()}>Logout</li>
                    </ul>
                  </div>

                  :

                  ""
                }
              </>
            }
         </>
       }
       </>
       }

     </>
   )
}

export default Navbar
```

Frontend > src > Components > common > PopularRestaurants.jsx

```
   import React, { useEffect,
useState } from 'react'
import
'../styles/PopularRestaurants.c
ss'
import { useNavigate } from
'react-router-dom';
import axios from 'axios';
```

```
const PopularRestaurants = ()
=> {

   const navigate =
useNavigate();

  const [restaurants,
setRestaurants] = useState([]);
const [promoteList,
setPromoteList] = useState([]);

   useEffect(()=>{
      fetchRestaurants();
      fetchPromotions();
    }, [])

   const  fetchRestaurants  =
async()       =>{
      await
axios.get('http://localhost:6001
/fetch-restaurants').then(
        (response)=>{

setRestaurants(response.data)
;
        }
      )
    }

    const  fetchPromotions  =
async () =>{
      await
axios.get('http://localhost:6001
/fetch-promoted-list').then(
        (response)=>{

setPromoteList(response.data)
;
        }
      )
    }



  return (
   <div
className="popularRestaura
ntContainer">
      <h3>Popular
Restaurants</h3>
```

```
      <div
className="popularRestaura
nt-body">


{restaurants.filter(restaurant=>
promoteList.includes(restaura
nt._id)).map((restaurant)=>(

            <div
className="popularRestaura
ntCard" key={restaurant._id}
onClick={()=>
navigate(`/restaurant/${restaur
ant._id}`)}>
              <img
src={restaurant.mainImg}
alt="" />
              <div
className="popularRestaura
ntCard-data">

<h6>{restaurant.title}</h6>

<p>{restaurant.address}</p>
              </div>
            </div>
         ))}


      </div>
    </div>
  )
}

export default
PopularRestaurants
```

Frontend > src > Components > common > Register.jsx

```
import React, { useContext, useState } from 'react'
import { GeneralContext } from
'../context/GeneralContext';

const Register = ({setIsLogin}) => {

  const {setUsername, setEmail, setPassword,
setUsertype, usertype, setRestaurantAddress,
setRestaurantImage, register} =
useContext(GeneralContext);
```

```
 const handleRegister = async (e) =>{
   e.preventDefault();
   await register();
 }


 return (
   <form className="authForm">
       <h2>Register</h2>
       <div className="form-floating mb-3
authFormInputs">
           <input type="text" className="form-control"
id="floatingInput" placeholder="username"
                                          onChange={(e)=>
setUsername(e.target.value)} />
           <label
htmlFor="floatingInput">Username</label>
       </div>
       <div className="form-floating mb-3
authFormInputs">
           <input type="email"
className="form-control" id="floatingEmail"
placeholder="name@example.com"
                                          onChange={(e)=>
setEmail(e.target.value)} />
           <label htmlFor="floatingInput">Email
address</label>
       </div>
       <div className="form-floating mb-3
authFormInputs">
           <input type="password"
className="form-control"  id="floatingPassword"
placeholder="Password"
                                          onChange={(e)=>
setPassword(e.target.value)} />
           <label
htmlFor="floatingPassword">Password</label>
       </div>
       <select className="form-select form-select-lg
mb-3" aria-label=".form-select-lg example"
                                         onChange={(e)=>
setUsertype(e.target.value)}>
         <option value="">User type</option>
         <option value="admin">Admin</option>
         <option
value="restaurant">Restaurant</option>
         <option value="customer">Customer</option>
       </select>

       {usertype === 'restaurant' ?
         <>
```

```
        <div className="form-floating mb-3
authFormInputs">
           <input type="text" className="form-control"
id="floatingAddress" placeholder="Address"
                                          onChange={(e)=>
setRestaurantAddress(e.target.value)} />
           <label
htmlFor="floatingAddress">Address</label>
        </div>
        <div className="form-floating mb-3
authFormInputs">
           <input type="text" className="form-control"
id="floatingImage" placeholder="Image"
                                          onChange={(e)=>
setRestaurantImage(e.target.value)} />
           <label htmlFor="floatingImage">Thumbnail
Image</label>
        </div>
        </>
       :
       ""
       }

      <button className="btn btn-primary"
onClick={handleRegister}>Sign up</button>
      <p>Already registered? <span onClick={()=>
setIsLogin(true)}>Login</span></p>
   </form>
  )}
```

Frontend > src > Components > admin >Restaurants.jsx

```
import React from 'react'
import '../styles/Restaurants.css'

const Restaurants = () => {
  return (
    <div className="restaurants-container">
       <div className="restaurants-filter">
          <h4>Filters</h4>
          <div className="restaurant-filters-body">

             <div className="filter-sort">
                <h6>Sort By</h6>
                <div className="filter-sort-body sub-filter-body">

                   <div className="form-check">
                      <input className="form-check-input" type="radio"
name="flexRadioDefault" id="filter-sort-radio1" />
                      <label className="form-check-label" htmlFor="filter-sort-radio1" >
                         Popularity
                      </label>
```

```
            </div>

            <div className="form-check">
              <input className="form-check-input" type="radio"
name="flexRadioDefault" id="filter-sort-radio4" />
              <label className="form-check-label" htmlFor="filter-sort-radio4">
                Rating
              </label>
            </div>

          </div>
        </div>
        <div className="filter-categories">
          <h6>Categories</h6>
          <div className="filter-categories-body sub-filter-body">

            <div className="form-check">
              <input className="form-check-input" type="checkbox" value=""
id="filter-category-check-1" />
              <label className="form-check-label"
htmlFor="filter-category-check-1">
                South Indian
              </label>
            </div>

            <div className="form-check">
              <input className="form-check-input" type="checkbox" value=""
id="filter-category-check-2" />
              <label className="form-check-label"
htmlFor="filter-category-check-2">
                North Indian
              </label>
            </div>

            <div className="form-check">
              <input className="form-check-input" type="checkbox" value=""
id="filter-category-check-3" />
              <label className="form-check-label"
htmlFor="filter-category-check-3">
                Chinese
              </label>
            </div>

            <div className="form-check">
              <input className="form-check-input" type="checkbox" value=""
id="filter-category-check-4" />
              <label className="form-check-label"
htmlFor="filter-category-check-4">
                Beverages
              </label>
            </div>
```

```
                  <div className="form-check">
                     <input className="form-check-input" type="checkbox" value=""
id="filter-category-check-5" />
                     <label className="form-check-label"
htmlFor="filter-category-check-5">
                        Ice Cream
                     </label>
                  </div>

                  <div className="form-check">
                     <input className="form-check-input" type="checkbox" value=""
id="flexCheckChecked" />
                     <label  className="form-check-label"  htmlFor="flexCheckChecked">
                        Tiffins
                     </label>
                  </div>
               </div>
            </div>

         </div>
      </div>



      <div className="restaurants-body">
         <h3>All restaurants</h3>
         <div className="restaurants">

            <div className='restaurant-item'>
               <div className="restaurant">
                  <img
src="https://odhi.in/image/cache/catalog/eat/chicken-biryani-odhi-in-eat-online-coim
batore-1000x1000.jpg" alt="" />
                  <div className="restaurant-data">
                     <h6>Product title</h6>
                     <p>Description about product</p>
                     <h5>Rating:  <b>3.6/5</b></h5>
                  </div>
               </div>
            </div>

            <div className='restaurant-item'>
               <div className="restaurant">
                  <img
src="https://odhi.in/image/cache/catalog/eat/chicken-biryani-odhi-in-eat-online-coim
batore-1000x1000.jpg" alt="" />
                  <div className="restaurant-data">
                     <h6>Product title</h6>
                     <p>Description about product</p>
                     <h5>Rating:  <b>3.6/5</b></h5>
                  </div>
```

```
          </div>
        </div>

        <div className='restaurant-item'>
          <div className="restaurant">
            <img
src="https://odhi.in/image/cache/catalog/eat/chicken-biryani-odhi-in-eat-online-coim
batore-1000x1000.jpg" alt="" />
            <div className="restaurant-data">
              <h6>Product title</h6>
              <p>Description about product</p>
              <h5>Rating: <b>3.6/5</b></h5>
            </div>
          </div>
        </div>

        <div className='restaurant-item'>
          <div className="restaurant">
            <img
src="https://odhi.in/image/cache/catalog/eat/chicken-biryani-odhi-in-eat-online-coim
batore-1000x1000.jpg" alt="" />
            <div className="restaurant-data">
              <h6>Product title</h6>
              <p>Description about product</p>
              <h5>Rating: <b>3.6/5</b></h5>
            </div>
          </div>
        </div>

        <div className='restaurant-item'>
          <div className="restaurant">
            <img
src="https://odhi.in/image/cache/catalog/eat/chicken-biryani-odhi-in-eat-online-coim
batore-1000x1000.jpg" alt="" />
            <div className="restaurant-data">
              <h6>Product title</h6>
              <p>Description about product</p>
              <h5>Rating: <b>3.6/5</b></h5>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
 )
}

export default Restaurants
```

12. AUTHENTICATION AND AUTHORIZATION:

1. Key Concepts

- Authentication: Verifying the identity of the user (e.g., login using credentials).
- Authorization: Granting or denying access to resources based on roles or permissions (e.g., admin vs customer).

2. Implementation Overview

1. Use JSON Web Tokens (JWT) for session-based authentication.
2. Define roles (e.g., admin, customer, delivery personnel) to handle authorization.
3. Secure sensitive endpoints using middleware.

3. Authentication

Steps to Implement

1. User Registration
   - Collect user details (e.g., name, email, password).
   - Hash passwords using bcrypt before storing them in the database.
   - Example with Mongoose:

```
javascript
Copy code
const bcrypt = require("bcrypt");
const UserSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  password: { type: String, required: true },
  role: { type: String, default: "customer" }, // Default role
});

UserSchema.pre("save", async function (next) {
  if (!this.isModified("password")) return next();
  this.password = await bcrypt.hash(this.password, 10);
  next();
});
module.exports = mongoose.model("User", UserSchema);
```

2. User Login
   - Validate email and password.
   - Generate a JWT token upon successful login.
   - Example:

```
javascript
Copy code
const jwt = require("jsonwebtoken");
const bcrypt = require("bcrypt");

const loginUser = async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  if (!user) return res.status(401).json({ message: "Invalid email or password" });

  const isMatch = await bcrypt.compare(password, user.password);
  if (!isMatch) return res.status(401).json({ message: "Invalid email or password"
```

3. Key Concepts

- Authentication: Verifying the identity of the user (e.g., login using credentials).
- Authorization: Granting or denying access to resources based on roles or permissions (e.g., admin vs customer).

4. Implementation Overview

4. Use JSON Web Tokens (JWT) for session-based authentication.
5. Define roles (e.g., admin, customer, delivery personnel) to handle authorization.
6. Secure sensitive endpoints using middleware.

3. Authentication

Steps to Implement

3. User Registration
   - Collect user details (e.g., name, email, password).
   - Hash passwords using bcrypt before storing them in the database.
   - Example with Mongoose:

     javascript
     Copy code
     ```
     const bcrypt = require("bcrypt");
     const UserSchema = new mongoose.Schema({
       name: String,
       email: { type: String, unique: true },
       password: { type: String, required: true },
       role: { type: String, default: "customer" }, // Default role
     });

     UserSchema.pre("save", async function (next) {
       if (!this.isModified("password")) return next();
       this.password = await bcrypt.hash(this.password, 10);
       next();
     });
     module.exports = mongoose.model("User", UserSchema);
     ```

4. User Login
   - Validate email and password.
   - Generate a JWT token upon successful login.
   - Example:

     javascript
     Copy code
     ```
     const jwt = require("jsonwebtoken");
     const bcrypt = require("bcrypt");

     const loginUser = async (req, res) => {
       const { email, password } = req.body;
       const user = await User.findOne({ email });
       if (!user) return res.status(401).json({ message: "Invalid email or password" });

       const isMatch = await bcrypt.compare(password, user.password);
       if (!isMatch) return res.status(401).json({ message: "Invalid email or password" });

       const token = jwt.sign({ id: user._id, role: user.role }, process.env.JWT_SECRET, { expiresIn:
     "1h" });
     ```

```
    res.json({ token });
  };
```

5. JWT Token
   - Include the JWT token in the response and save it on the frontend (e.g., local storage or cookies).
   - Middleware to validate tokens for protected routes.

6. Authorization

Role-Based Access Control (RBAC)

1. Define roles for users:
   - Admin: Manage users, restaurants, and orders.
   - Customer: Place orders, view their history.
   - Delivery Personnel: Access assigned delivery orders.
2. Use middleware to check permissions:

```
javascript
Copy code
const authorize = (roles) => {
  return (req, res, next) => {
    const userRole = req.user.role; // Assuming user info is attached to the request
    if (!roles.includes(userRole)) {
      return res.status(403).json({ message: "Access denied" });
    }
    next();
  };
};

// Example usage
app.get("/admin/dashboard", authenticateToken, authorize(["admin"]), (req, res) => {
  res.send("Welcome to the Admin Dashboard");
});
```

5. Securing Endpoints

1. Authentication Middleware
   - Validate the token and attach user information to the request.

```
javascript
Copy code
const authenticateToken = (req, res, next) => {
  const token = req.headers["authorization"]?.split(" ")[1];
  if (!token) return res.status(401).json({ message: "Access denied" });

  jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
    if (err) return res.status(403).json({ message: "Invalid token" });
    req.user = user;
    next();
  });
};
```

2. Protect Routes
   - Apply authenticateToken to routes that require authentication.

```
javascript
Copy code
app.get("/orders", authenticateToken, (req, res) => {
  // Fetch user-specific orders
});
```

3. Role-Based Routes
   - Combine authenticateToken with authorize middleware to restrict access.

```
javascript
Copy code
app.post("/restaurants", authenticateToken, authorize(["admin"]), (req, res) => {
  // Admin-specific functionality
});
```

6. Example Workflow

Customer Use Case

1. A new user registers via /api/auth/register.
2. The user logs in via /api/auth/login and receives a JWT token.
3. The user places an order via /api/orders, passing the JWT token in the Authorization header.

Admin Use Case

1. An admin logs in and receives a JWT token.
2. The admin accesses **/api/admin/users** to manage users.
3. The admin can only perform admin-level operations due to role-based middleware.

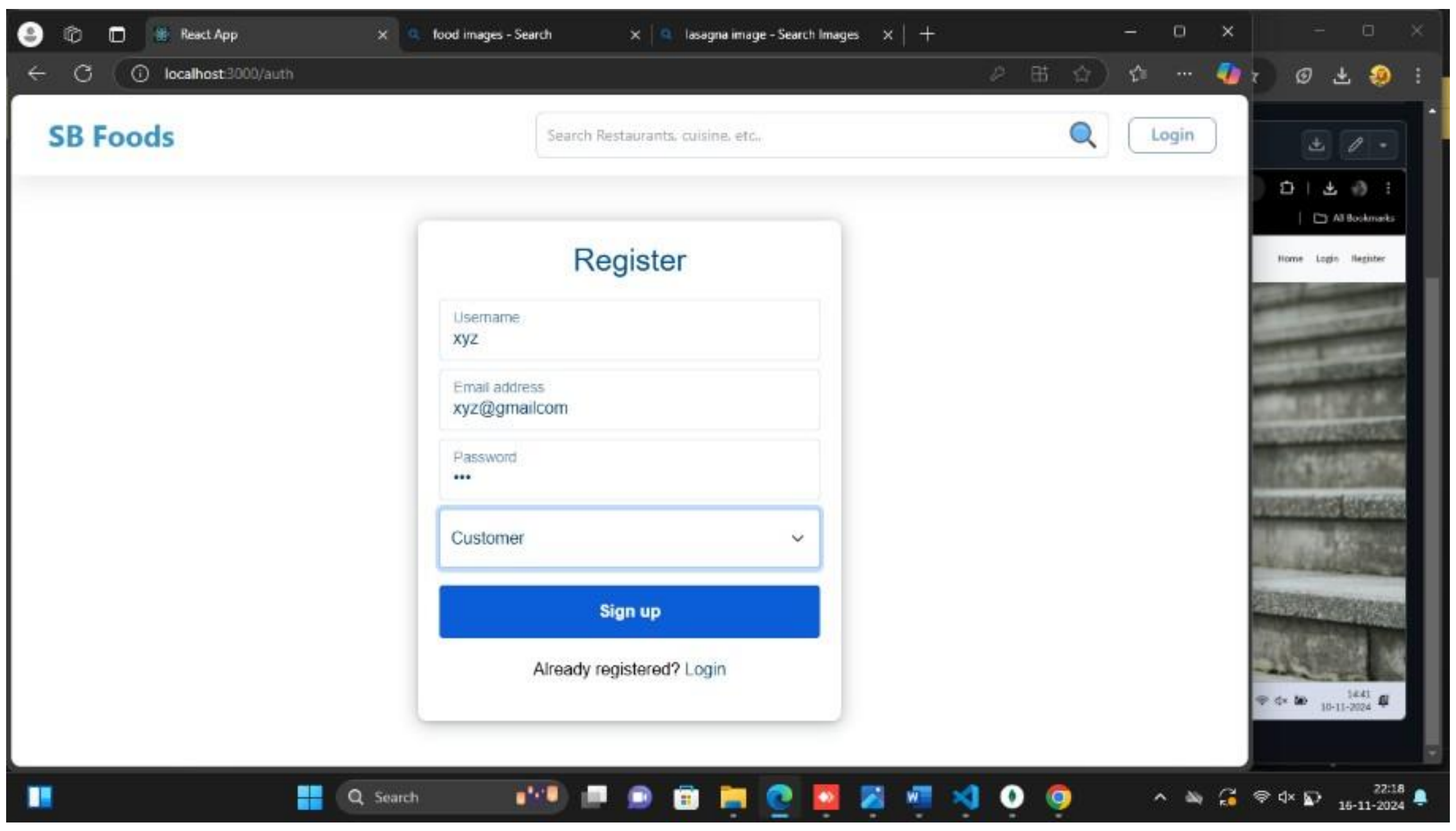
7. Security Best Practices

1. Use HTTPS to secure communication.
2. Store secrets (e.g., **JWT_SECRET**) in environment variables.
3. Implement token expiration to prevent long-term abuse.
4. Use refresh tokens for extended sessions.
5. Prevent Cross-Site Scripting (XSS) by sanitizing inputs.
6. Implement rate-limiting to prevent brute force attacks.

8 Example .env File

```
env
Copy code
PORT=5000
MONGO_URI=mongodb://localhost:27017/sbfoodapp
JWT_SECRET=your_jwt_secret_key
```

SB Food App


React App
food images - Search
lasagna image - Search Images
localhost:3000/auth
SB Foods
Search Restaurants, cuisine, etc..
Login

Register

Username
xyz

Email address
xyz@gmailcom

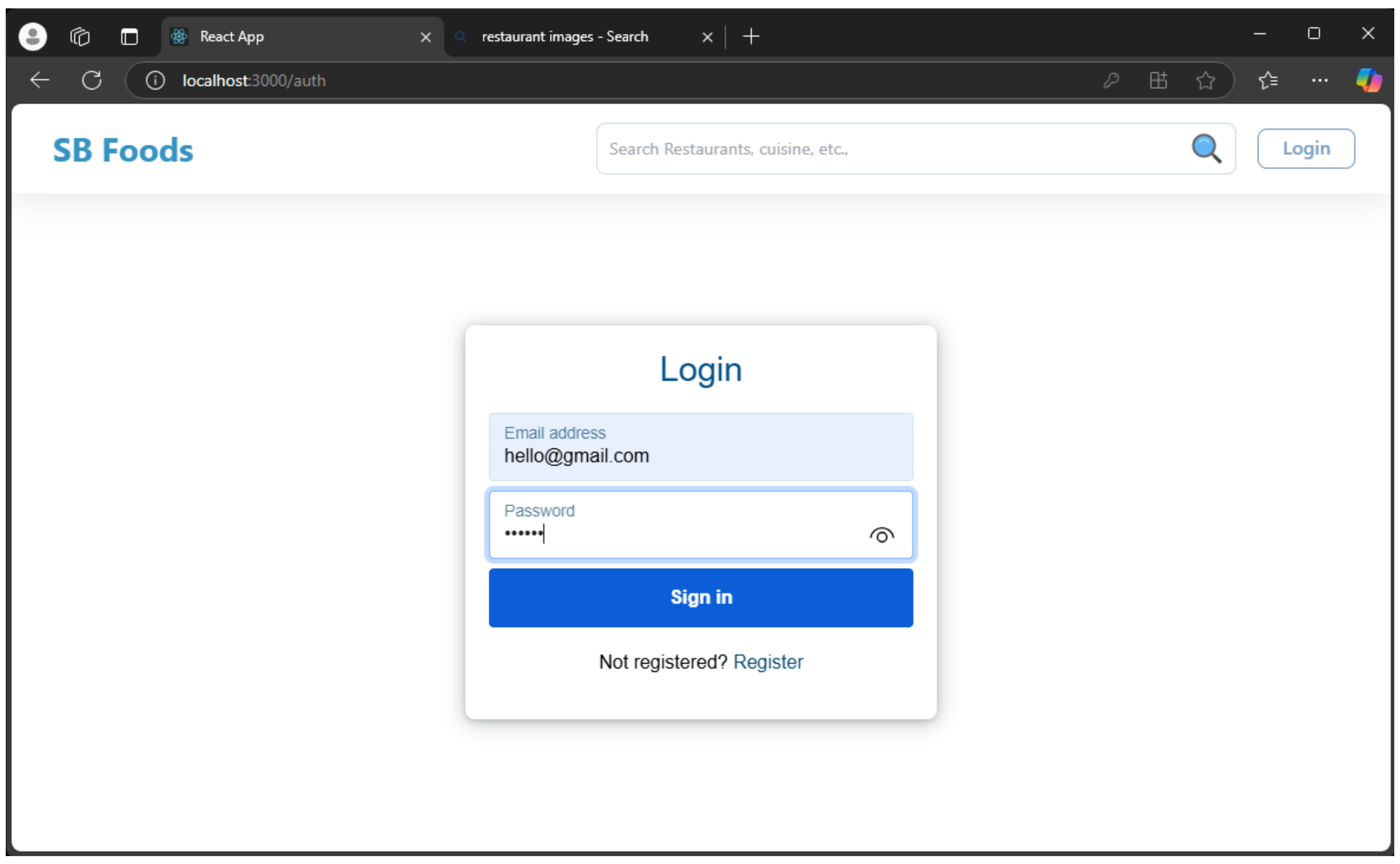
Password
•••

Customer

Sign up

Already registered? Login

SB Food App Login Page


React App
restaurant images - Search
localhost:3000/auth
SB Foods
Search Restaurants, cuisine, etc..
Login

Login

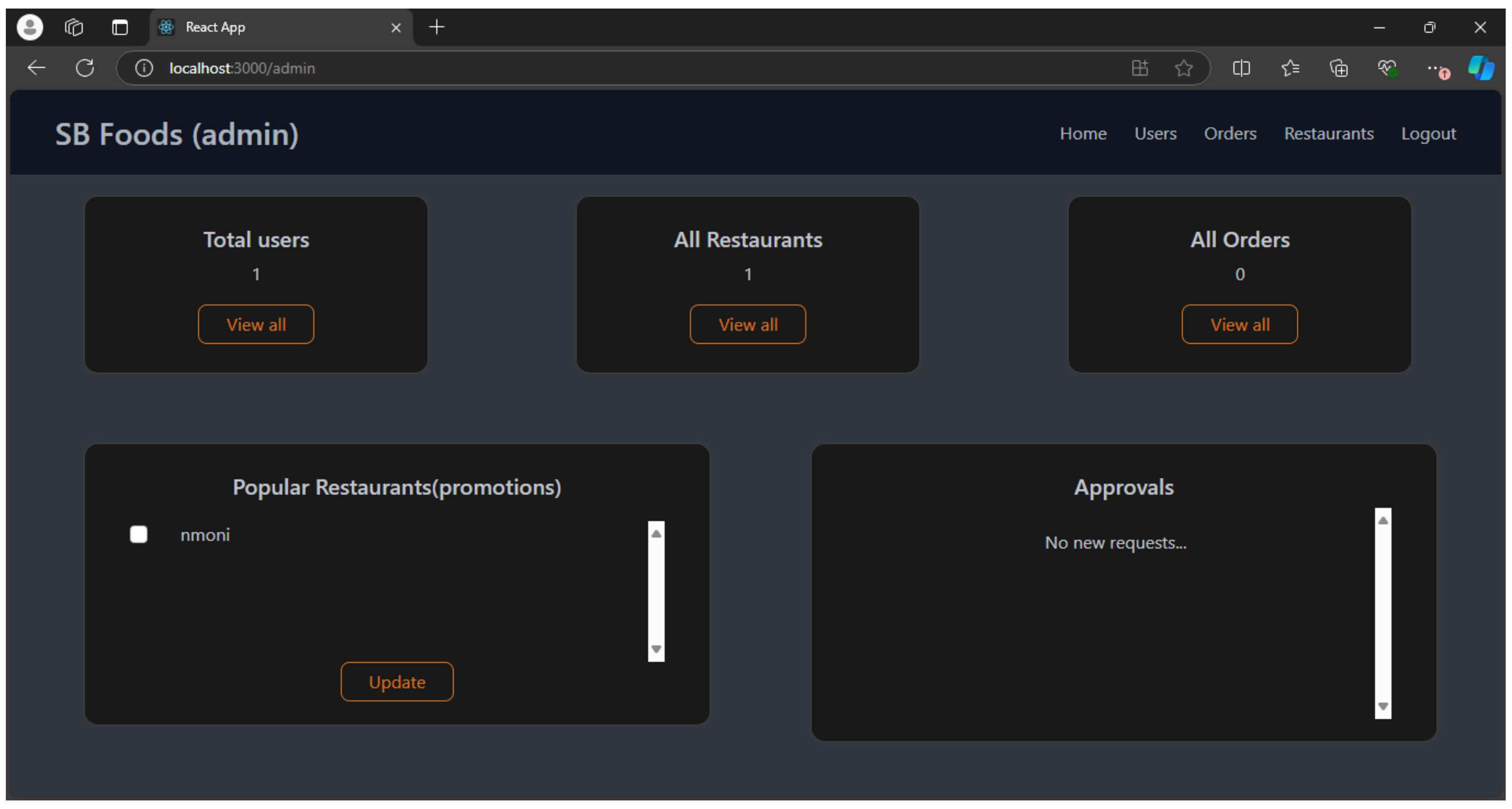
Email address
hello@gmail.com

Password
••••••

Sign in

Not registered? Register

SB Food App Home Page



Total users
1
View all

All Restaurants
1
View all

All Orders
0
View all

Popular Restaurants(promotions)

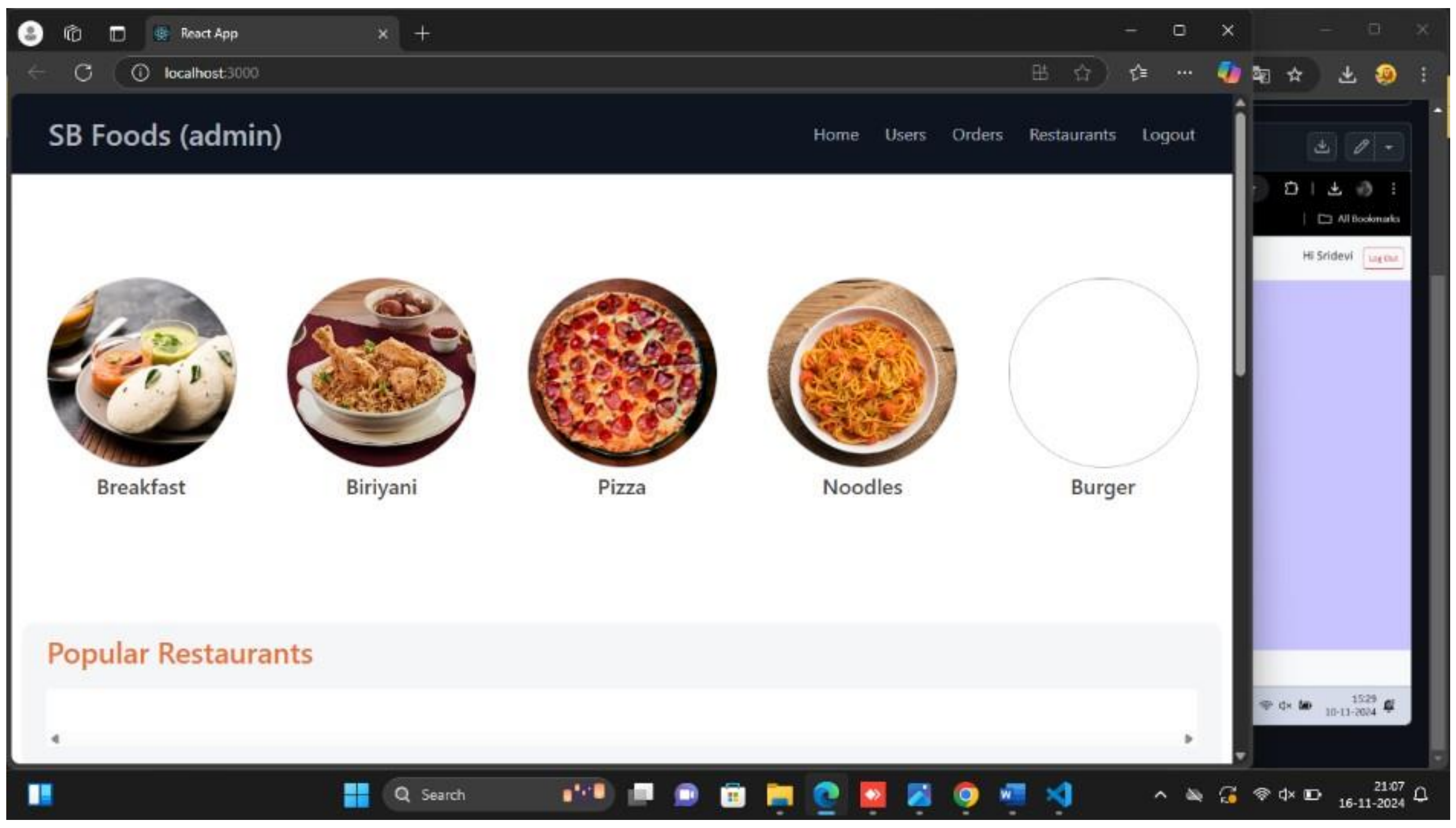
nmoni

Update

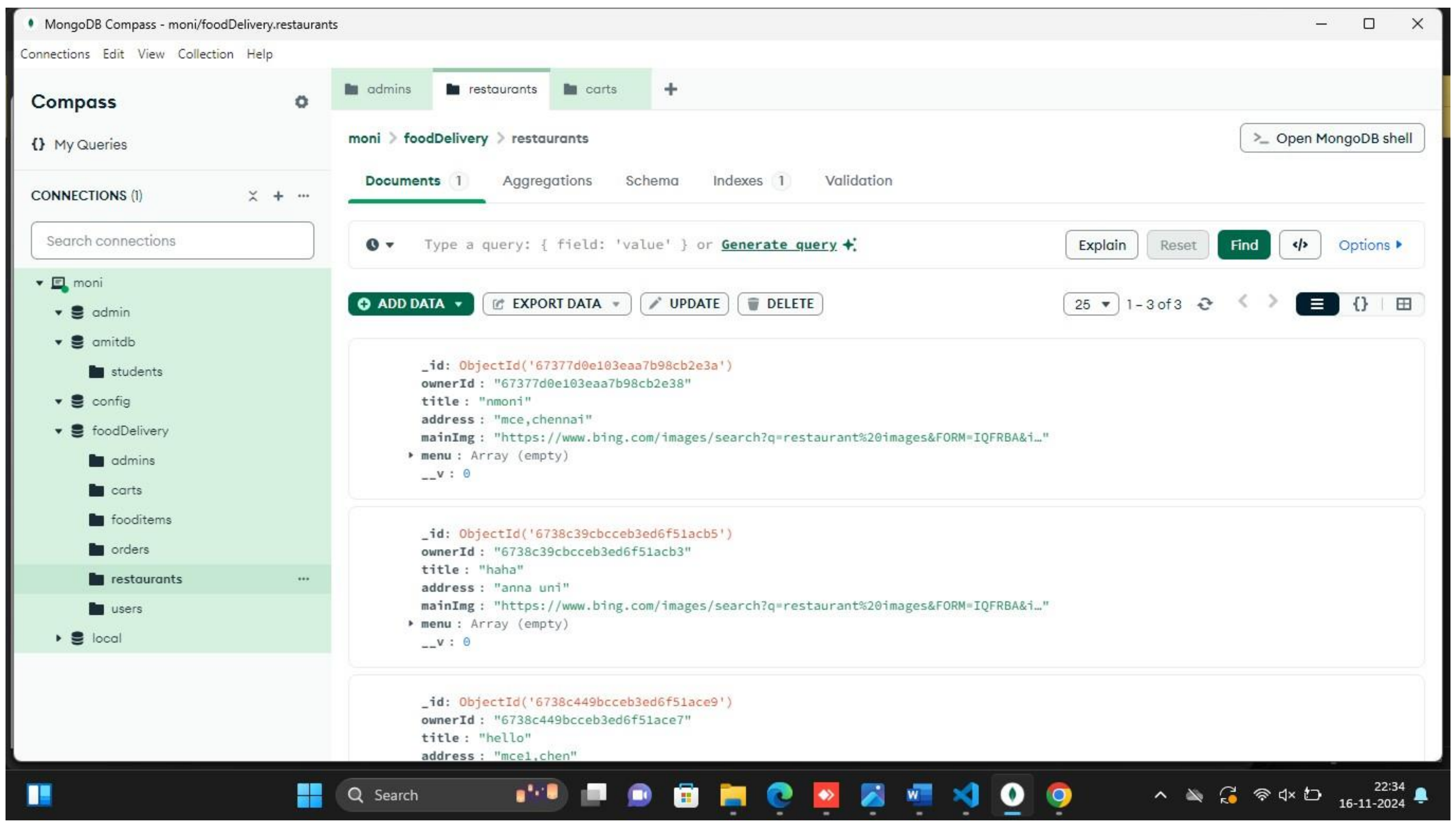
Approvals

No new requests...

Items added to homepage



Restaurant Database

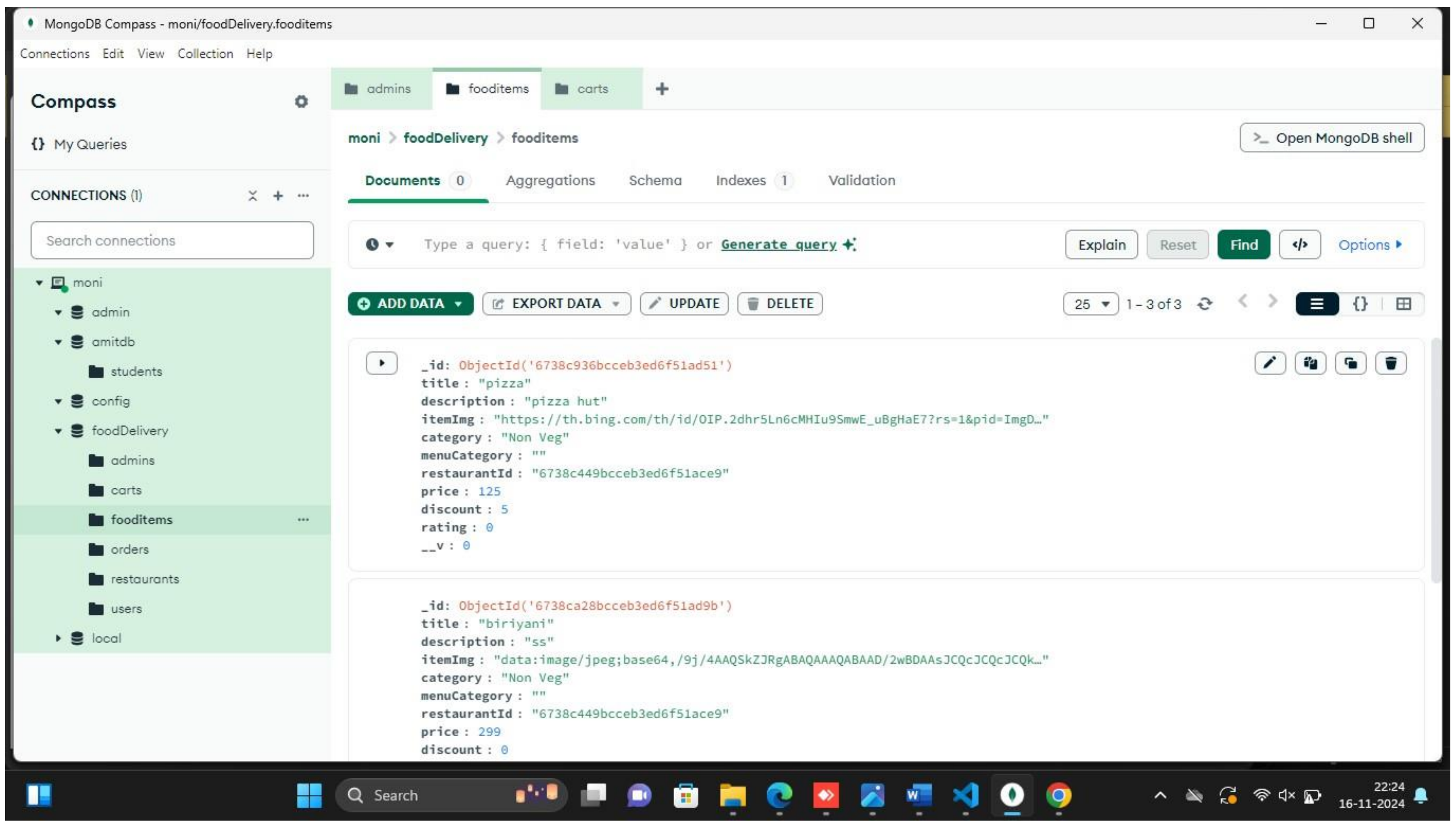


```
_id: ObjectId('67377d0e103eaa7b98cb2e3a')
ownerId : "67377d0e103eaa7b98cb2e38"
title : "nmoni"
address : "mce,chennai"
mainImg : "https://www.bing.com/images/search?q=restaurant%20images&FORM=IQFRBA&i…"
menu : Array (empty)
__v : 0

_id: ObjectId('6738c39cbcceb3ed6f51acb5')
ownerId : "6738c39cbcceb3ed6f51acb3"
title : "haha"
address : "anna uni"
mainImg : "https://www.bing.com/images/search?q=restaurant%20images&FORM=IQFRBA&i…"
menu : Array (empty)
__v : 0

_id: ObjectId('6738c449bcceb3ed6f51ace9')
ownerId : "6738c449bcceb3ed6f51ace7"
title : "hello"
address : "mce1,chen"
```

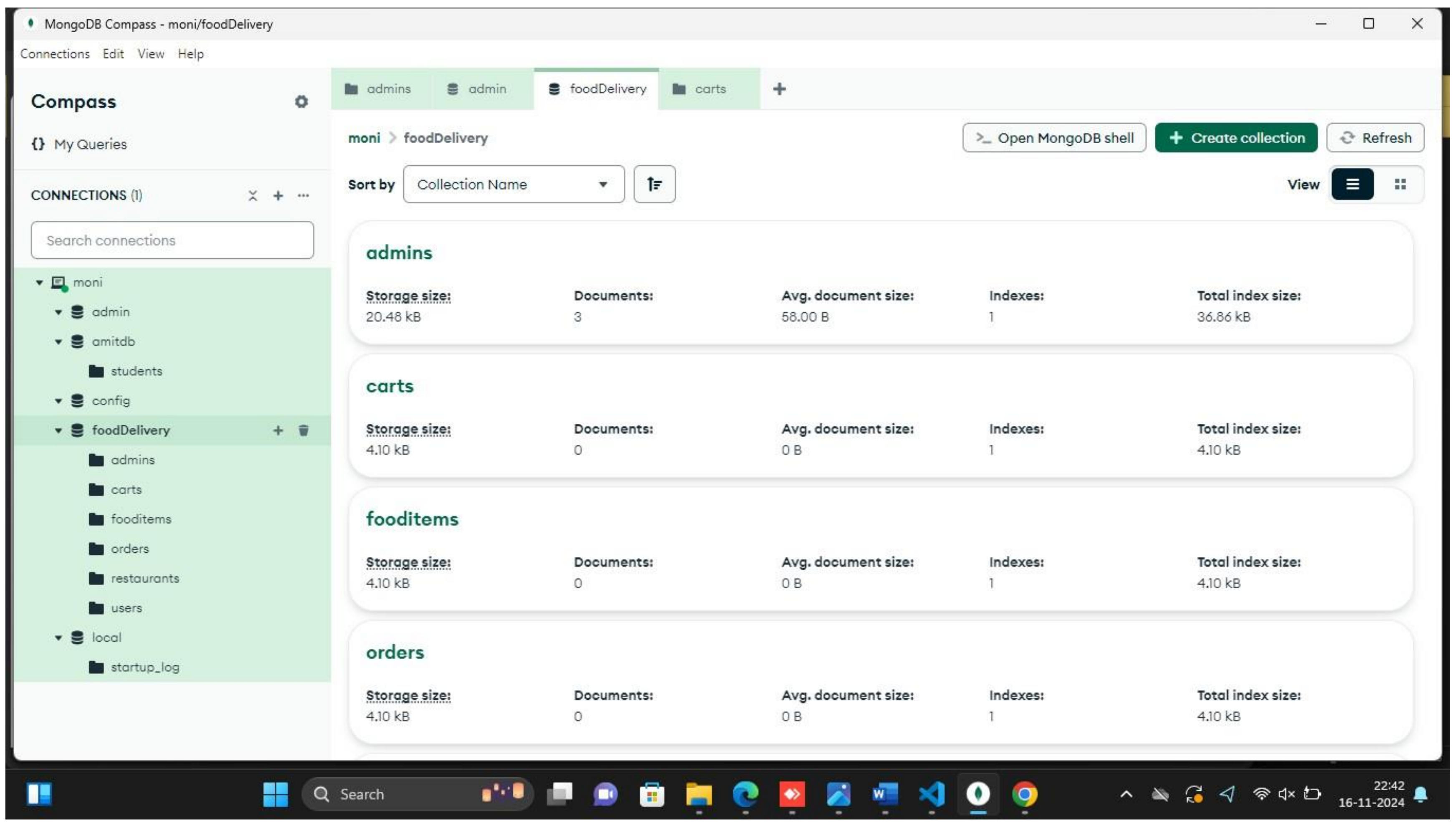
Food Item Database



```
_id: ObjectId('6738c936bcceb3ed6f51ad51')
title : "pizza"
description : "pizza hut"
itemImg : "https://th.bing.com/th/id/OIP.2dhr5Ln6cMHIu9SmwE_uBgHaE7?rs=1&pid=ImgD…"
category : "Non Veg"
menuCategory : ""
restaurantId : "6738c449bcceb3ed6f51ace9"
price : 125
discount : 5
rating : 0
__v : 0

_id: ObjectId('6738ca28bcceb3ed6f51ad9b')
title : "biriyani"
description : "ss"
itemImg : "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAAQABAAD/2wBDAAsJCQcJCQcJCQk…"
category : "Non Veg"
menuCategory : ""
restaurantId : "6738c449bcceb3ed6f51ace9"
price : 299
discount : 0
```

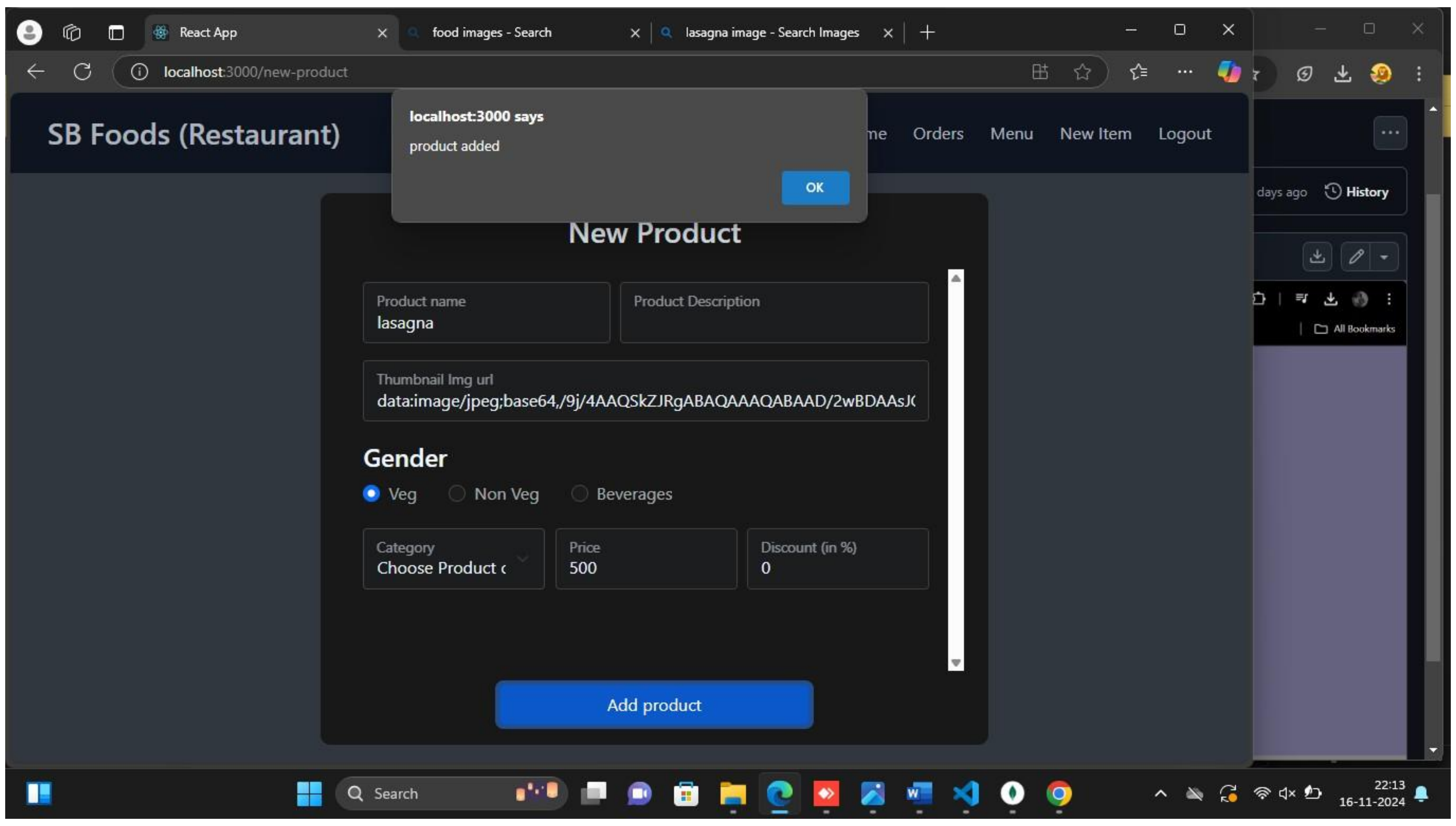
Collection Database



moni > foodDelivery

	Storage size:	Documents:	Avg. document size:	Indexes:	Total index size:
admins	20.48 kB	3	58.00 B	1	36.86 kB
carts	4.10 kB	0	0 B	1	4.10 kB
fooditems	4.10 kB	0	0 B	1	4.10 kB
orders	4.10 kB	0	0 B	1	4.10 kB

New Item Added



localhost:3000/new-product

localhost:3000 says
product added

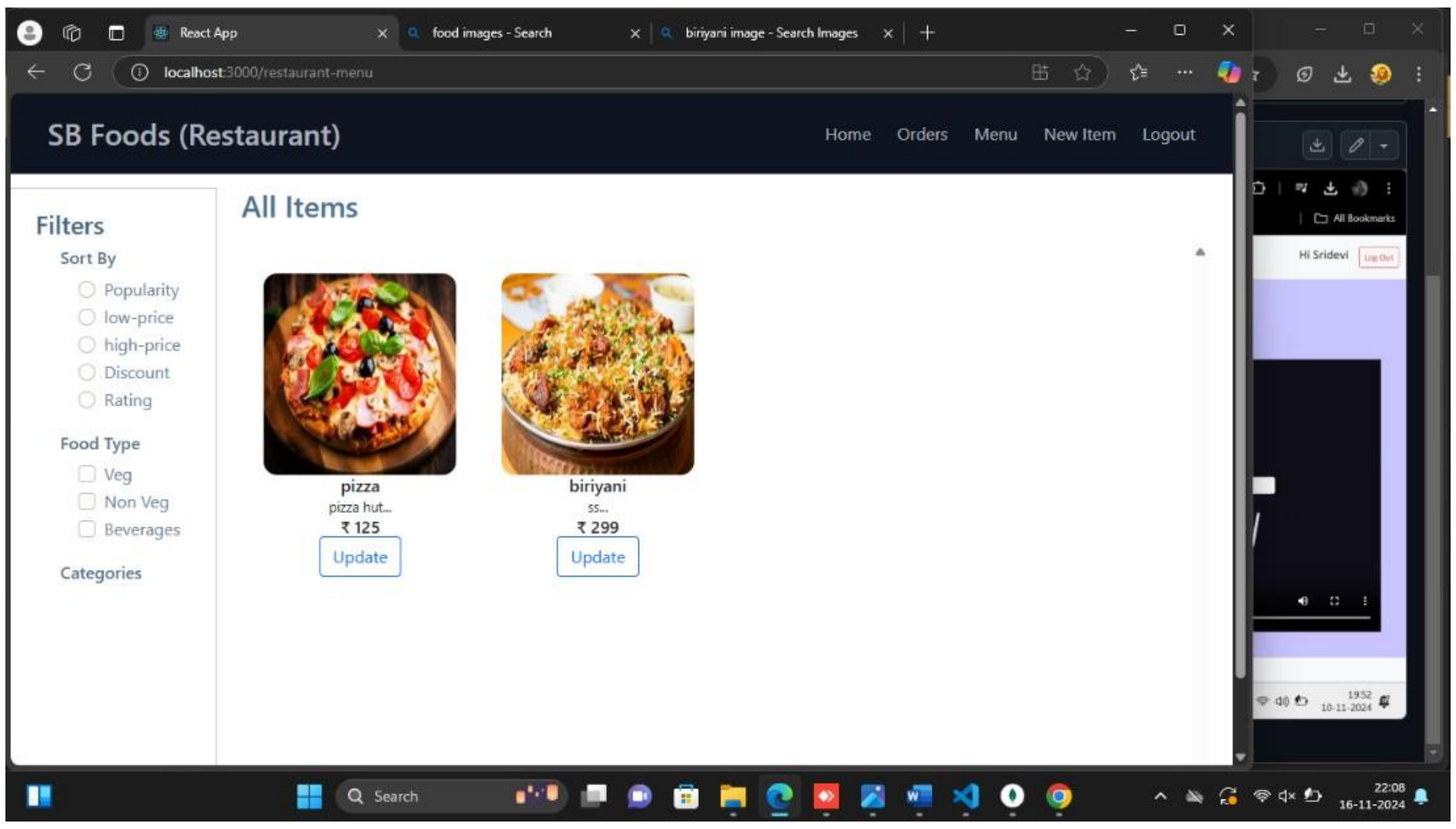
New Product

Product name: lasagna
Product Description
Thumbnail Img url: data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAAQABAAD/2wBDAAsJ

Gender
[✓ Veg] Non Veg Beverages

Category: Choose Product
Price: 500
Discount (in %): 0

Users Card Items



localhost:3000/restaurant-menu

All Items

pizza
pizza hut...
₹ 125

biriyani
ss...
₹ 299

13. TESTING:

Testing Strategy and Tools for the Online Learning Platform (OLP)

1. Testing Strategy Overview

The testing strategy for SB Food App will cover unit testing, integration testing, end-to-end (E2E) testing, API testing, and UI testing.

- Unit Testing: Testing individual components, services, or functions in isolation.
- Integration Testing: Testing interactions between multiple components or modules (e.g., database integration, external services).
- End-to-End (E2E) Testing: Simulating user behavior and testing the app as a whole from frontend to backend.
- API Testing: Ensuring the API endpoints return the expected responses and handle edge cases correctly.
- UI Testing: Ensuring the frontend works properly and UI components function as expected.

2. Types of Testing

a.Unit Testing

Unit tests verify the functionality of individual units of code such as functions, methods, or components. These are typically run in isolation from the rest of the application.

- Frontend:
  - Test React components to verify their behavior.
  - Mock dependencies (e.g., API calls, third-party libraries).
  - Use Jest for testing and React Testing Library for DOM interaction testing.
- Backend:
  - Test individual utility functions, controllers, or middleware.
  - Use Mocha and Chai for unit testing, with Sinon for mocking.

Tools for Unit Testing:

- Frontend:
  - Jest: JavaScript testing framework.
  - React Testing Library: For testing React components.
  - Enzyme: A utility for testing React components (optional, less common now).
- Backend:
  - Mocha: Test framework for Node.js.
  - Chai: Assertion library for Mocha.
  - Sinon: For mocking and spying on functions.

b. Integration Testing

Integration tests ensure that different modules of the app work together as expected. These tests are critical for verifying interactions between the frontend, backend, and database.

- Frontend:
  - Test API calls, interactions between components, and component integration.
  - Jest with React Testing Library can also be used for integration testing, especially when combined with mock services.
- Backend:
  - Test database integration, API routes, and service layers.
  - Use Mocha/Chai for testing and mock database interactions.

Tools for Integration Testing:

- Frontend:
  - Jest + React Testing Library.
  - Cypress (for end-to-end tests that include API interaction).
- Backend:
  - Supertest: For HTTP assertion testing.
  - Mocha/Chai: For backend service integration.

c. End-to-End (E2E) Testing

End-to-end testing simulates real user behavior, testing the app from the frontend all the way through to the backend, including database interactions.

- Tools:
  - Cypress: For simulating real user interactions, including frontend and API.
  - Playwright: Another modern tool for end-to-end testing.
  - Selenium (alternative): Older tool for automated browser testing.

E2E Testing Strategy:

- Test user workflows such as login, order placement, and payment.
- Simulate user behaviors (e.g., navigating the app, submitting forms, and verifying data updates).
-
  - Test individual utility functions, controllers, or middleware.
  - Use Mocha and Chai for unit testing, with Sinon for mocking.

Tools for Unit Testing:

- Frontend:
  - Jest: JavaScript testing framework.
  - React Testing Library: For testing React components.
  - Enzyme: A utility for testing React components (optional, less common now).
- Backend:
  - Mocha: Test framework for Node.js.
  - Chai: Assertion library for Mocha.
  - Sinon: For mocking and spying on functions.

d. Integration Testing

Integration tests ensure that different modules of the app work together as expected. These tests are critical for verifying interactions between the frontend, backend, and database.

- Frontend:
  - Test API calls, interactions between components, and component integration.
  - Jest with React Testing Library can also be used for integration testing, especially when combined with mock services.
- Backend:
  - Test database integration, API routes, and service layers.
  - Use Mocha/Chai for testing and mock database interactions.

Tools for Integration Testing:

- Frontend:
  - Jest + React Testing Library.
  - Cypress (for end-to-end tests that include API interaction).
- Backend:
  - Supertest: For HTTP assertion testing.
  - Mocha/Chai: For backend service integration.

e. End-to-End (E2E) Testing

End-to-end testing simulates real user behavior, testing the app from the frontend all the way through to the backend, including database interactions.

- Tools:
  - Cypress: For simulating real user interactions, including frontend and API.
  - Playwright: Another modern tool for end-to-end testing.
  - Selenium (alternative): Older tool for automated browser testing.

E2E Testing Strategy:

- Test user workflows such as login, order placement, and payment.
- Simulate user behaviors (e.g., navigating the app, submitting forms, and verifying data updates).

f. API Testing

API testing ensures that the backend routes and business logic behave as expected. Testing focuses on HTTP methods (GET, POST, PUT, DELETE) and the data they handle.

- Tools:
  - Postman: For manual API testing and automated API testing through collections.
  - Supertest: For automated testing of API routes directly from the backend.

API Testing Strategy:

- Test endpoints with different request methods (GET, POST, PUT, DELETE).
- Validate request body, query parameters, headers, and responses.
- Test edge cases, invalid inputs, and authorization failures.

Example (API Testing with Postman):

- Create a Postman Collection for your API endpoints.
- Test different routes such as `/login`, `/orders`, and `/restaurants`.

g. UI Testing

UI testing ensures that the user interface is working as expected, looking and functioning correctly on different devices and screen sizes.

- Tools:
  - Cypress: Can also be used for testing the frontend UI and user interactions.
  - Selenium: For cross-browser testing of UI components.
  - Puppeteer: For headless browser testing (automating Chrome).

4. Continuous Integration and Automation

To ensure consistent testing and quick feedback, integrate tests with a Continuous Integration (CI) pipeline.

- CI Tools:
  - Jenkins
  - GitHub Actions
  - CircleCI
  - Travis CI
- Automation Steps:
  - Run unit tests, integration tests, and API tests on every commit and pull request.
  - Automate E2E tests on staging/deployment environments.

15. KNOWN ISSUES:

1. Frontend Issues

1.1. UI Layout Issues

- Issue: Some UI elements might not align correctly on different screen sizes or browsers.
  - Impact: Affects the user experience on mobile devices or specific browsers (e.g., Internet Explorer).
  - Solution: Use responsive design frameworks (e.g., Bootstrap, Tailwind CSS) and test on multiple devices.

1.2. Form Validation

- Issue: Form validation is inconsistent, especially for user inputs such as email or password fields.
  - Impact: Users may not be able to submit forms correctly if validation fails.
  - Solution: Implement thorough client-side validation with libraries like Formik or React Hook Form.

1.3. Slow Page Load Times

- Issue: Some pages (e.g., menu, order details) load slowly due to large image files

or non-optimized assets.
  - Impact: Decreases user experience, leading to potential abandonment.
  - Solution: Optimize images using tools like ImageOptim or TinyPNG, and implement lazy loading for heavy assets.

1.4. Unresponsive Buttons

- Issue: Buttons or actions may not trigger events on mobile devices due to improper handling of touch events.
  - Impact: Users may not be able to interact with key elements.
  - Solution: Test button actions on all devices, and ensure event handlers are optimized for both desktop and mobile.

1.5. Cross-Browser Compatibility

- Issue: Some browsers (e.g., Safari, Edge) may not render the app's features correctly.
  - Impact: Certain UI components may look broken or behave incorrectly on non-Chrome browsers.
  - Solution: Use Autoprefixer and test the app on multiple browsers.

2. Backend Issues

2.1. JWT Token Expiration

- Issue: JWT tokens may expire too quickly, causing users to get logged out unexpectedly.
  - Impact: Users may have a poor experience if they need to log in again frequently.
  - Solution: Increase token expiration duration and implement refresh tokens to extend user sessions.

2.2. Database Connection Issues

- Issue: MongoDB or other database connections might intermittently fail due to network issues or incorrect configurations.
  - Impact: The app may fail to fetch or store data, leading to broken functionalities like menu fetching or order placement.
  - Solution: Use database connection pools and implement retry logic for failed connections.

2.3. Slow API Responses

- Issue: Some API routes (e.g., fetching large orders) may take longer than expected to return data.
  - Impact: Affects the user experience by causing delays in fetching data.
  - Solution: Optimize database queries, implement pagination, and cache frequent queries.

2.4. Insufficient Authorization Checks

- Issue: In some cases, authorization checks might be missing or insufficient,

allowing unauthorized users to access admin routes or other restricted areas.
  - Impact: Security vulnerabilities could expose sensitive information.
  - Solution: Ensure role-based access control is implemented correctly and test a l sensitive routes for proper access control.

3. Performance Issues

3.1. High Server Load During Peak Traffic

- Issue: The app might experience slowdowns or crashes during high traffic times (e. g., lunch or dinner hours).
  - Impact: Performance degradation leading to poor user experience and potential service downtime.

  Solution: Scale the backend servers, implement load balancing, and use a Content Delivery Network (CDN) to serve static content.

3.2. Image and File Upload Delays

- Issue: Uploading images or other files might take too long due to unoptimized file sizes or slow server configurations.
  - Impact: Delays in order placements or profile updates.
  - Solution: Optimize file uploads, allow for asynchronous uploads, and consider integrating a cloud-based storage service like AWS S3.

4. User Management Issues

4.1. Account Deletion

- Issue: Account deletion may not fu ly remove all user data from the database.
  - Impact: Users may sti l see their information after deleting their accounts, leading to privacy concerns.
  - Solution: Implement proper data deletion logic in the backend that clears all user-related data when requested.

4.2. Password Reset Issues

- Issue: Password reset emails may fail to send or contain incorrect links due to email service misconfigurations.
  - Impact: Users will be unable to reset their passwords, which can lock them out of their accounts.
  - Solution: Test email configurations, ensure correct token generation, and use reliable services like SendGrid for sending reset emails.

5. 5.1. Payment Gateway

Integration

- Issue: Integration with payment gateways (e.g., Stripe, PayPal) might occasionally

fail or show incorrect error messages.
  - Impact: Users cannot complete their orders or payments.
  - Solution: Test payment gateway integrations thoroughly, and ensure error handling is clear and provides appropriate feedback.

5.2. Incorrect Order Pricing

- Issue: Orders may be priced incorrectly due to bugs in the pricing logic or promotional discount calculation.
  - Impact: Users could be charged the wrong amount, which might lead to complaints or abandoned orders.
  - Solution: Ensure a l pricing logic is thoroughly tested and validated. Implement unit tests for discount and price calculations.

Miscellaneous Issues

5.1. Localization and Language Issues

- Issue: Some parts of the app might not be properly translated, or the localization might break on certain screens.
  - Impact: Non-English speaking users may face difficulties using the app.
  - Solution: Implement a comprehensive localization strategy using tools like i18next and ensure all strings are correctly translated.

5.2. Push Notifications Not Working

- Issue: Push notifications may fail to trigger or show incorrect messages due to issues with Firebase or another notification service.
  - Impact: Users may miss important updates like order status or promotional offers.
  - Solution: Ensure push notification services are properly configured, and test thoroughly on multiple devices.

6. Known Edge Cases

7.1. High Order Volume

- Issue: The system might not handle a high volume of orders efficiently, leading to delayed processing.
  - Impact: Orders may be delayed or missed entirely.
  - Solution: Implement queuing systems (e.g.,RabbitMQ) to handle orders efficiently and scale backend services accordingly.

7.2. Multiple Devices Logging In

- Issue: Users might experience issues when logging in on multiple devices (e.g., conflicting session states

18. CONCLUSION:

The SB Food App is designed to offer a seamless and user-friendly

experience for bothcustomers and restaurant partners. By providing an intuitive platform for browsing menus,placing orders, and making payments, it aims to enhance the food delivery experience.
Key features such as user authentication, real-time order tracking, payment gateway integration, and restaurant management make it a comprehensive solution for modernfood delivery needs.

While the development of the app includes robust testing, including unit, integration, andend-to-end tests, certain known issues such as UI responsiveness, payment gatewayintegration, and database connectivity must be addressed to ensure optimal performance. The app is built with scalability in mind, ensuring it can handle a growing user base and increased traffic, especially during peak hours.

The use of modern technologies, such as React for the frontend, Node.js for the backend,and MongoDB for database management, ensures that the app is both reliable and easy to maintain. Moreover, implementing security measures like JWT-based authentication and role-based authorization ensures that user data and interactions are we l protected.

In conclusion, the SB Food App is poised to provide a seamless, efficient, and secure food delivery experience. By addressing the known issues and continuously improving the app based on user feedback, it can become a trusted platform for food ordering and delivery, convenienceand reliability for customers and restaurant owners alike.

GitHub Project Link: https:/ github.com/suryakala10/SB-FOOD-APP